6

Mekanizma: Sınırlı Doğrudan Yürütme

Bir CPU'yu sanallaştırmak için, işletim sistemi fiziksel CPU'yu birçok aynı anda çalışan iş arasında nasıl paylaşabilecek. Temel fikir basittir: bir işlemi biraz süre için çalıştırın, sonra diğerini çalıştırın ve bu şekilde devam edin. CPU'yu bu şekilde **zaman paylaşarak(time sharing),** sanallaştırma gerçekleştirilir. Bununla birlikte, böyle bir sanallaştırma makinesi inşa etmek için birkaç zorluk vardır. İlki performans: sisteme aşırı yük eklemeden nasıl sanallaştırma gerçekleştirebiliriz? İkinci olarak, kontrol: CPU'da kontrolü korumayı nasıl sağlayarak işlemleri verimli bir şekilde çalıştırabiliriz? Kontrol, OS için özellikle önemlidir, çünkü kaynakları yönetir; kontrol olmadan, bir işlem sürekli çalışarak makineyi ele geçirebilir veya erişmesine izin verilmeyen bilgilere erişebilir. Yüksek performansı korurken kontrol sağlamak, işletim sistemi inşa etmenin ana zorluklarından biridir.

İşin püf noktası:

CPU'YU CONTROL ILE VERIMLI BIR ŞEKILDE SANALLAŞTIRMA
İşletim sistemi, sistem üzerindeki kontrolü korurken CPU'yu verimli bir şekilde sanallaştırmalıdır. Bunu yapmak için hem donanım hem de işletim sistemi desteği gerekecektir. İşletim sistemi, işini etkili bir şekilde gerçekleştirmek için genellikle makul bir donanım desteği kullanacaktır.

6.1 Temel Teknik: Sınırlı Doğrudan Yürütme

Beklendiği kadar hızlı çalışan bir program oluşturmak için, şaşırtıcı değil ki işletim sistemi geliştiricileri, **"sınırlı doğrudan çalıştırma(Limited Direct Execution)"** olarak adlandırdığımız bir teknik geliştirdiler. Fikrin "doğrudan çalıştırma" bölümü basittir: sadece programı CPU üzerinde doğrudan çalıştırın. Bu nedenle, işletim sistemi bir programı çalıştırmayı istediğinde, bir işlem listesinde bir işlem girişi oluşturur, bazı bellek ayırır, kodu belleğe yükler (diskten), giriş noktasını bulur (yani ana () rutini veya benzeri bir şey), atlar ve kodu çalıştırmaya başlar.

OS	Program
İşlem listesi için giriş oluştur	
Program için bellek ayırma	
Programı belleğe yükle argc/argv ile yığını temizle	
Execute call main()	
	Ana () çalıştır
	Ana sayfadan
	dönüş(Return)
	yürütme
İşlemin boş belleği	
İşlem listesinden kaldırın	

Şekil 6.1: Doğrudan Yürütme Protokolü (Sınırsız)

tıklayın ve kullanıcının kodunu çalıştırmaya başlayın. Şekil 6.1, bu temel di-rect yürütme protokolünü (henüz herhangi bir sınır olmadan), normal bir çağrı kullanarak ve programın ana () öğesine ve daha sonra çekirdeğe geri dönmek için gösterir.

Kulağa basit geliyor, değil mi? Ancak bu yaklaşım, CPU'yu sanallaştırma arayışımızda birkaç soruna yol açıyor. Birincisi basittir: sadece bir programı çalıştırırsak, işletim sistemi programın yapmasını istemediğimiz hiçbir şeyi yapmadığından nasıl emin olabilir? İkincisi: Bir işlemi çalıştırırken, işletim sistemi çalışmasını nasıl durdurur ve başka bir işleme geçer, böylece CPU'yu sanallaştırmak için ihtiyaç duyduğumuz **zaman paylaşımını(Time sharing)** nasıl uygular?

Aşağıdaki soruları cevaplarken, CPU'yu sanallaştırmak için neyin gerekli olduğuna dair çok daha iyi bir fikir edineceğiz. Bu teknikleri geliştirirken, ismin "sınırlı" kısmının nereden geldiğini de göreceğiz; programları çalıştırma konusunda sınırlamalar olmadan, işletim sistemi hiçbir şeyin kontrolünde olmazdı ve bu nedenle "sadece bir kütüphane" olurdu - hevesli bir işletim sistemi için çok üzücü bir durum!

6.1 Sorun #1: Kısıtlı İşlemler

Doğrudan yürütme, hızlı olmanın bariz avantajına sahiptir; program donanım CPU'sunda yerel olarak çalışır ve böylece beklendiği kadar hızlı çalışır. Ancak CPU üzerinde çalışmak bir soruna neden olur: İşlem bir diske G/Ç isteği vermek veya CPU veya bellek gibi daha fazla sistem kaynağına erişmek gibi bir tür kısıtlı işlem gerçekleştirmek isterse ne olur?

İŞIN PÜF NOKTASI: KISITLI IŞLEMLER NASIL GERÇEKLEŞTIRILIR

Bir süreç, G/Ç ve diğer bazı kısıtlı işlemleri gerçekleştirebilmelidir, ancak sürece sistem üzerinde tam kontrol sağlamadan. İşletim sistemi ve donanım bunu yapmak için birlikte nasıl çalışabilir?

SİSTEM ÇAĞRILARI NEDEN BİR YÖNTEM ÇAĞRISI GİBİ GÖRÜNÜR

open() veya read() gibi bir sistem çağrısına bir çağrının neden C'de tipik bir yöntem çağrısı gibi göründüğünü merak edebilirsiniz; yani, bir yöntem çağrısı gibi görünüyorsa, sistem nasıl bildiğini ve doğru şeyleri yaptığını merak edebilirsiniz? Basit bir neden: bir yöntem çağrısıdır, ancak içinde ünlü bir kapanma talimatı vardır. Daha spesifik olarak, open () çağrısı yaptığınızda (örneğin), C kütüphanesine bir yöntem çağrısı yapıyorsunuz. İçinde, open () veya diğer sistem çağrıları için, kütüphane ile çekirdek arasında bir anlaşma kullanarak open () için argümanları belli yerlere koyar (örneğin, yığına veya belirli bir kaydedicilere), sistem çağrı numarasını da belirli bir yere koyar (yine, yığına veya bir kaydediciye) ve daha sonra ünlü kapanma talimatını yürütür. Kütüphanedeki kod kapanma sonrası dönüş değerlerini açar ve sistem çağrısı yapılan programa kontrolü geri verir. Bu nedenle, sistem çağrısı yapan C kütüphanesinin bölümleri elle kodlanmıştır, çünkü argümanları ve dönüş değerlerini doğru bir şekilde işlemek ve ayrıca donanım-özel kapanma talimatını yürütmek için dikkatli bir şekilde anlaşmaya uymalıdır. Ve şimdi neden kişisel olarak bir işletim sistemine kapanma için derleme kodu yazmanıza gerek yok; biri zaten o derlemi sizin için yazdı.

Bir yöntem, I/O ve diğer ilgili işlemler açısından herhangi bir işlemi istediği şekilde yapmasına izin vermek olurdu. Ancak böyle yapmak, pek çok istenen tür sistemlerin inşasını engelleyecektir. Örneğin, bir dosya sistemi inşa etmek istiyorsunuz ve bir dosyaya erişim izni vermeden önce izinleri kontrol etmek istiyorsanız, kullanıcı işlemlerine disk üzerinde I/O emri vermemek zorundayız; eğer yapsaydık, bir işlem basitçe diskte okuyabilir veya yazabilir ve böylece tüm korunmalar kaybolurdu.

Bu nedenle, aldığımız yöntem, **kullanıcı modu(User mode)** olarak bilinen yeni bir işlemci modu eklemektir; kullanıcı modunda çalışan kodun yapabileceği şeyler sınırlıdır. Örneğin, kullanıcı modunda çalışırken, bir işlem I/O isteği yapamaz; bunu yapmak işlemcinin bir istisna oluşturmasına neden olur; işletim sistemi daha sonra işlemi öldürmeyi düşünebilir.

Kullanıcı moduna karşılık gelen **kernel modu(kernel mode**), işletim sistemi (veya çekirdek) tarafından çalıştırılır. Bu modda, çalışan kodun sevdiği şeyleri yapabilir, I/O istekleri yapma ve tüm kısıtlı talimatları yürütme gibi özel işlemleri de içerebilir. Yine de bir zorlukla karşı karşıyayız: Bir kullanıcı işlemi, diskten okuma gibi bir özel işlemi gerçekleştirmek istediğinde ne yapmalıdır? Bunu sağlamak için, hemen hemen tüm modern donanım, kullanıcı programlarının bir **sistem çağrısı(system call)** yapmasına izin verme yeteneğine sahiptir.

Eski makinelerde (Atlas [K + 61, L78] gibi) ortaya çıkarılan sistem çağrıları, çekirdeğin kullanıcı programlarına dosya sistemine erişim, işlem oluş

İPUCU: KORUMLU KONTROL TRANSFER KULLANIN

Donanım, işletim sistemine farklı çalışma modları sağlar. **Kullanıcı modunda(user mode)**, uygulamalar donanım kaynaklarına tam erişim hakkına sahip değildir. **Kernel modunda(kernel mode),** işletim sistemi makineye tam kaynaklara erişebilir.Çekirdeğe ve kullanıcı modundaki programlardan **dönüş yapma(return-from trap)** için **kapama(trap)** talimatlarının yanı sıra, işletim sisteminin donanıma **kapanma tablosunun(trap table)** bellekte bulunduğu yeri bildirebilen talimatlar da sağlanır.

Bellek. Çoğu işletim sistemi birkaç yüz çağrı sağlar (ayrıntılar için POSIX standardına bakın [P10]); erken Unix sistemleri yirmiden fazla çağrıdan oluşan daha kapsamlı bir alt kümesini açıkladı. Bir sistem çağrısını yürütmek için, bir programın bir özel **kapanma(trap)** talimatını yürütmesi gerekir. Bu talimat aynı anda çekirdeğe atlar ve yetki düzeyini kernel moduna yükseltir; bir kez çekirdekte, sistem şimdi izin verilen (eğer izin verilirse) gerekli olan özel işlemleri gerçekleştirebilir ve böylece çağrı yapan işlem için gereken işi yapabilir. Tamamlandığında, OS bir özel **dönüş(return-from trap)** çağırır ve beklediğiniz gibi, çağrı yapan kullanıcı programına geri döner ve aynı zamanda yetki düzeyini kullanıcı moduna düşürür.

Kapama yapılırken, çağrının kaydedilen birkaç kaydının yeterince kaydedilmiş olması gerektiğinden donanım biraz dikkatli olmalıdır. Örneğin, x86 için, işlemci çağrının program sayacını, bayrakları ve birkaç diğer kaydı işlem başına bir **kernel yığınına(Kernel stack**) itecektir; return-from-trap bu değerleri yığından pop edecek ve kullanıcı modundaki programın çalışmasına devam edecektir (ayrıntılar için Intel sistemleri el kitabına bakın [I11]). Diğer donanım sistemleri farklı kurallara sahip olabilir ancak temel kavramlar platformlar arasında benzerdir.

Bu tartışmanın dışında kalan önemli bir ayrıntı var: Tuzak, işletim sistemi içinde hangi kodun çalıştırılacağını nasıl biliyor? Açıkçası, arama işlemi atlamak için bir adres belirtemez (prosedürlü bir arama yaparken yaptığınız gibi); bunu yapmak, programların açıkça **Çok Kötü Bir Fikir(Very bad idea)** olan çekirdeğe herhangi bir yere atlamasına izin verecektir. Bu nedenle çekirdek, bir tuzak üzerinde hangi kodun yürütüldüğünü dikkatlice kontrol etmelidir.

Çekirdek bunu, önyükleme sırasında bir tuzak tablosu(trap table) ayarlayarak yapar. Makine önyüklendiğinde, bunu ayrıcalıklı (çekirdek) modda yapar ve böylece makine donanımını gerektiği gibi yapılandırmakta özgürdür. İşletim sisteminin bu nedenle yaptığı ilk şeylerden biri, donanıma belirli istisnai olaylar meydana geldiğinde hangi kodun çalıştırılacağını söylemektir. Örneğin, bir sabit disk kesintisi gerçekleştiğinde, klavye kesintisi gerçekleştiğinde veya bir program sistem çağrısı yaptığında hangi kod çalışmalıdır? İşletim sistemi, donanımı bilgilendirir

1Bir dosyaya erişmek için koda atladığınızı, ancak izin kontrolünden hemen sonra olduğunu hayal edin; Aslında, böyle bir yeteneğin kurnaz bir programcının çekirdeğin rasgele kod dizileri çalıştırmasını sağlaması muhtemeldir [S07]. Genel olarak, bunun gibi Çok Kötü Fikirlerden kaçınmaya çalışın.

OS @ boot (kernel mode)	Hardware	
kapanma tablosunu başlat		
	adresini hatırla... syscall işleyicisi	
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
İşlem listesi için giriş oluştur		
Program için bellek ayır		
Programı belleğe yükle argv ile kullanıcı yığınını kurun		
Reg/PC return from-trap ile çekirdek yığınını doldurun		
	regs geri yükleme (çekirdek yığınından) kullanıcı moduna geçin ana moda atla	
		kayıtları kaydet (çekirdek yığınına) çekirdek moduna geçin tuzak işleyicisine atla
	regs geri yükleme (çekirdek yığınından) kullanıcı moduna geçin Tuzaktan sonra PC'ye geçin	
Sap tuzağı		
Syscall işini yapın		
tuzaktan dönüş		
	Ana () çalıştır	
	...	
	Çağrı sistemi çağrısı işletim sistemine tuzak	
	...	
	ana sayfadan dönüş	
İşlemin boş belleği işlem listesinden kaldırın	tuzak (çıkış yoluyla())	

Figure 6.2: Limited Direct Execution Protocol

Bu **kapanma kısımlarının(trap handlers)** yerlerini genelde bazı özel talimatlarla bildirir. Donanım bilgilendirildikten sonra, bu kısımların yerlerini, makine bir kez daha yeniden başlatılana kadar hatırlar ve böylece donanım sistem çağrıları ve diğer olağanüstü olaylar gerçekleştiğinde ne yapılacağını (yani nereye atlaması gerektiğini) bilir.

İPUCU: GÜVENLI SISTEMLERDEKI KULLANICI GIRIŞLERINE KARŞI DIKKATLI OLUN

Sistem çağrıları sırasında işletim sistemini korumak için büyük çaba sarf etmiş olsak da (bir donanım yakalama mekanizması ekleyerek ve işletim sistemine yapılan tüm çağrıların üzerinden yönlendirilmesini sağlayarak), güvenli bir işletim sistemini uygulamak için göz önünde bulundurmamız gereken birçok başka yön vardır. Bunlardan biri, sistem çağrı sınırındaki argümanların ele alınmasıdır; işletim sistemi, kullanıcının neyi geçtiğini kontrol etmeli ve bağımsız değişkenlerin doğru bir şekilde belirtildiğinden emin olmalı veya çağrıyı başka bir şekilde reddetmelidir.

Örneğin, write() sistem çağrısında, kullanıcı yazma çağrısının kaynağı olarak arabelleğin adresini belirtir. Kullanıcı (yanlışlıkla veya kötü niyetle) "kötü" bir adrese geçerse (örneğin, çekirdeğin adres alanının bir kısmı içinde), işletim sistemi bunu algılamalı ve çağrıyı reddetmelidir. Aksi takdirde, bir kullanıcının tüm çekirdek belleğini okuması mümkün olacaktır; Çekirdek (sanal) belleğin genellikle sistemin tüm fiziksel belleğini de içerdiği göz önüne alındığında, bu küçük kayma bir programın sistemdeki diğer işlemlerin belleğini okumasını sağlar.

Genel olarak, güvenli bir sistem kullanıcı girdilerine büyük şüpheyle yaklaşmalıdır. Bunu yapmamak şüphesiz kolayca saldırıya uğrayan yazılımlara, dünyanın güvensiz ve korkutucu bir yer olduğuna dair umutsuz bir duyguya ve çok güvenen işletim sistemi geliştiricisi için iş güvenliği kaybına yol açacaktır.

Tam sistem çağrısını belirtmek için, genellikle her sistem çağrısına bir **sistem çağrısı numarası(system call number)** atanır. Bu nedenle kullanıcı kodu, istenen sistem çağrı numarasını bir kasaya veya yığın üzerinde belirli bir yere yerleştirmekten sorumludur; işletim sistemi, tuzak işleyicisinin içindeki sistem çağrısını işlerken, bu numarayı inceler, geçerli olduğundan emin olur ve eğer öyleyse, korelasyon kodunu yürütür. Bu yönlendirme seviyesi bir **koruma(protection)** biçimi olarak hizmet eder; kullanıcı kodu, atlamak için tam bir adres belirtemez, bunun yerine numara aracılığıyla belirli bir hizmeti talep etmelidir.

Son bir kenara: tuzak masalarının nerede olduğunu sert donanıma söyleme talimatını yerine getirebilmek çok güçlü bir yetenektir. Dolayısıyla tahmin edebileceğiniz gibi **ayrıcalıklı(privileged)** bir işlemdir. Bu talimatı kullanıcı modunda yürütmeye çalışırsanız, donanım size izin vermez ve muhtemelen ne olacağını tahmin edebilirsiniz (ipucu: adios, rahatsız edici program). Düşünmek için bir nokta: Kendi tuzak masanızı kurabilseydiniz bir sisteme ne gibi korkunç şeyler yapabilirdiniz? Makineyi devralabilir misin?

Zaman çizelgesi (Şekil 6.2'de zaman aşağı doğru artarken) protokolü özetler. Her işlemin, çekirdeğe girip çıkarken reg- ister'lerin (genel amaçlı kayıtlar ve program sayacı dahil) kaydedildiği ve (donanım tarafından) geri yüklendiği bir çekirdek yığınına sahip olduğunu varsayıyoruz.

Sınırlı doğrudan yürütme (**LDE**) protokolünde iki aşama vardır. İlk (önyükleme zamanında), çekirdek tuzak tablosunu başlatır ve CPU daha sonra kullanmak üzere konumunu hatırlar. Çekirdek bunu ayrıcalıklı bir talimatla yapar (tüm ayrıcalıklı talimatlar kalın harflerle vurgulanır).

İkincisinde (bir işlemi çalıştırırken), çekirdek, işlemin yürütülmesini başlatmak için bir tuzaktan dönüş talimatı vermeden önce birkaç şey (örneğin, işlem listesinde bir düğüm ayırmak, bellek ayırmak) kurar; bu, CPU'yu kullanıcı moduna geçirir ve işlemi çalıştırmaya başlar. İşlem bir sistem çağrısı yapmak istediğinde, işletim sistemine geri döner, bu da onu ele alır ve bir kez daha tuzaktan geri dönüş yoluyla kontrolü işleme geri döndürür. İşlem daha sonra çalışmasını tamamlar ve main(); bu genellikle programdan düzgün bir şekilde çıkacak bazı saplama koduna geri döner (örneğin, işletim sistemine hapsolan exit() sistem çağrısını çağırarak). Bu noktada, işletim sistemi temizlenir ve işimiz biter.

6.2 Sorun #2: İşlemler Arasında Geçiş Yapma

Doğrudan yürütme ile ilgili bir sonraki sorun, süreçler arasında bir geçiş yapmaktır. Süreçler arasında geçiş yapmak basit olmalı, değil mi? İşletim sistemi sadece bir işlemi durdurmaya ve başka bir işlemi başlatmaya karar vermelidir. Önemli olan nedir? Ancak aslında biraz zor: özellikle, CPU üzerinde bir işlem çalışıyorsa, bu tanım gereği işletim sisteminin çalışmadığı anlamına gelir. İşletim sistemi çalışmıyorsa, nasıl bir şey yapabilir? (ipucu: yapamaz) Bu neredeyse felsefi gibi görünse de, gerçek bir sorundur: CPU'da çalışmıyorsa işletim sisteminin bir işlem yapmasının bir yolu yoktur. Böylece sorunun özüne ulaşmış oluruz.

IŞIN PÜF NOKTASI: CPU'NUN KONTROLÜ NASIL YENIDEN KAZANILIR

İşletim sistemi, işlemler arasında geçiş yapabilmesi için CPU'nun kontrolünü nasıl yeniden ele geçirebilir?

İşbirlikçi Bir Yaklaşım: Sistem Çağrılarını Bekleyin

Bazı sistemlerin geçmişte benimsediği bir yaklaşım (örneğin, Macintosh işletim sisteminin [M11] veya eski Xerox Alto sisteminin [A79]) ilk sürümleri **işbirlikçi(cooperative)** yaklaşım olarak bilinir. Bu tarzda, işletim sistemi sistemin süreçlerinin makul davranacağına güvenir. Çok uzun süre çalışan işlemlerin, işletim sisteminin başka bir görevi çalıştırmaya karar verebilmesi için CPU'dan periyodik olarak vazgeçtiği varsayılır.

Bu nedenle, dostça bir süreç bu ütopik dünyada CPU'dan nasıl vazgeçer? Çoğu işlem, ortaya çıktığı gibi, örneğin bir dosyayı açmak ve daha sonra okumak veya başka bir makineye mesaj göndermek veya yeni bir işlem oluşturmak için **sistem çağrıları(system call)** yaparak CPU'nun kontrolünü işletim sistemine oldukça sık aktarır. Bunun gibi sistemler genellikle kontrolü işletim sistemine aktarmaktan başka bir şey yapmayan açık bir **verim(yield)** sistemi çağrısı içerir, böylece diğer işlemleri çalıştırabilir.
ayrıca yasadışı bir şey yaptıklarında kontrolü işletim sistemine aktarırlar.
Örneğin, bir uygulama sıfıra bölünürse veya erişememesi gereken belleğe erişmeye çalışırsa,

İşletim sistemi. İşletim sistemi daha sonra CPU'nun kontrolünü tekrar ele geçirecek (ve muhtemelen rahatsız edici işlemi sonlandıracaktır).
Böylece, işbirlikçi bir zamanlama sisteminde, işletim sistemi bir sistem çağrısının veya bir tür yasadışı işlemin gerçekleşmesini bekleyerek CPU'nun kontrolünü yeniden kazanır. Ayrıca şunu da düşünüyor olabilirsiniz: Bu pasif yaklaşım idealden daha az değil mi? Örneğin, bir süreç (kötü amaçlı veya sadece hatalarla dolu) sonsuz bir döngüde sona ererse ve asla bir sistem çağrısı yapmazsa ne olur? İşletim sistemi o zaman ne yapabilir?

İşbirlikçi Olmayan Bir Yaklaşım: İşletim Sistemi Kontrolü Ele Alıyor

Donanımdan bazı ek yardımlar olmadan, bir işlem sistem çağrıları (veya hataları) yapmayı reddettiğinde ve böylece kontrolü işletim sistemine geri döndürdüğünde işletim sisteminin hiç bir şey yapamayacağı ortaya çıkıyor. Aslında, işbirlikçi yaklaşımda, bir süreç sonsuz bir döngüde sıkışıp kaldığında tek başvurunuz, bilgisayar sistemlerindeki tüm sorunlara asırlık çözüme başvurmaktır: **makineyi yeniden başlatın(reboot the machine).** Böylece, CPU'nun kontrolünü ele geçirmek için genel arayışımızın bir alt sorununa tekrar varıyoruz.

İŞIN PÜF NOKTASI: İŞBIRLIĞI OLMADAN KONTROL NASIL KAZANILIR

İşletim sistemi, işlemler işbirliği yapmasa bile CPU'nun kontrolünü nasıl ele geçirebilir? İşletim sistemi, hileli bir işlemin makineyi ele geçirmemesini sağlamak için ne yapabilir?

Cevabının basit olduğu ortaya çıktı ve yıllar önce bilgisayar sistemleri inşa eden birkaç kişi tarafından keşfedildi: **bir zamanlayıcı kesintisi(timer interrupt)** [M + 63]. Bir zamanlayıcı cihazı, her milisaniyede bir kesinti oluşturacak şekilde programlanabilir; kesme oluşturulduğunda, çalışmakta olan işlem durdurulur ve işletim sistemindeki önceden yapılandırılmış bir **kesme işleyicisi(interrupt handler)** çalışır. Bu noktada, işletim sistemi CPU'nun kontrolünü yeniden ele geçirdi ve böylece istediğini yapabilir: mevcut işlemi durdurun ve farklı bir işlem başlatın.

Daha önce sistem çağrılarıyla tartıştığımız gibi, işletim sistemi, zamanlayıcı kesintisi gerçekleştiğinde hangi kodun çalıştırılacağı konusunda donanımı bilgilendirmelidir; Böylece, önyükleme zamanında, işletim sistemi tam olarak bunu yapar. İkincisi, önyükleme sırasında da, işletim sistemi elbette ayrıcalıklı bir zamanlayıcı başlatmalıdır.

İPUCU: UYGULAMANIN YANLIŞ DAVRANIŞLARIYLA BAŞA ÇIKMA

İşletim sistemleri genellikle tasarım (kötü niyetlilik) veya kaza (hatalar) yoluyla yapmamaları gereken bir şey yapmaya çalışan yanlış çalışan süreçlerle uğraşmak zorunda kalır. Modern sistemlerde, işletim sisteminin bu tür bir kötüye kullanımı ele almaya çalışmasının yolu, suçluyu basitçe feshetmektir. Bir grev ve sen dışarıdasın! Belki acımasız, ancak belleğe yasadışı olarak erişmeye çalıştığınızda veya yasadışı bir talimatı yerine getirdiğinizde işletim sistemi başka ne yapmalı?

işlem. Zamanlayıcı başladıktan sonra, işletim sistemi bu nedenle kontrolün sonunda kendisine geri döneceği konusunda güvende hissedebilir ve böylece işletim sistemi kullanıcı programlarını çalıştırmakta özgürdür. Zamanlayıcı da kapatılabilir (aynı zamanda ayrıcalıklı bir işlem), daha sonra eşzamanlılığı daha ayrıntılı olarak anladığımızda tartışacağımız bir şey.

Bir kesme gerçekleştiğinde, özellikle de kesinti gerçekleştiğinde çalışmakta olan programın durumundan yeterince tasarruf etmek için donanımın bazı sorumlulukları olduğunu unutmayın, böylece sonraki bir tuzaktan dönüş yönergesi çalışan programı doğru şekilde sürdürebilir. Bu eylemler kümesi, donanımın çekirdeğe açık bir sistem çağrısı tuzağı sırasındaki davranışına oldukça benzer, böylece çeşitli kayıtlar kaydedilir (örneğin, bir çekirdek yığınına) ve böylece tuzaktan dönüş talimatıyla kolayca geri yüklenir.

Bağlamı Kaydetme ve Geri Yükleme

Artık işletim sistemi, ister bir sistem çağrısı yoluyla işbirliği içinde ister bir zamanlayıcı kesintisi yoluyla daha güçlü bir şekilde olsun, kontrolü yeniden ele geçirdiğine göre, bir karar verilmelidir: şu anda çalışan işlemi çalıştırmaya devam etmek veya farklı bir işleme geçmek. Bu karar, işletim sisteminin **zamanlayıcı(scheduler)** olarak bilinen bir bölümü tarafından verilir; Önümüzdeki birkaç bölümde zamanlama politikalarını ayrıntılı olarak tartışacağız.

Geçiş kararı verilirse, işletim sistemi daha sonra **bağlam anahtarı(context switch)** olarak adlandırdığımız düşük seviyeli bir kod parçası yürütür. Bir bağlam anahtarı kavramsal olarak basittir: işletim sisteminin tek yapması gereken, şu anda yürütülmekte olan işlem için birkaç kayıt değeri kaydetmek (örneğin, çekirdek yığınına) ve yakında yürütülecek işlem için birkaçını geri yüklemektir (çekirdek yığınından). Bunu yaparak, işletim sistemi böylece tuzaktan dönüş talimatı nihayet yürütüldüğünde, çalışan işleme geri dönmek yerine, sistemin başka bir işlemin yürütülmesine devam etmesini sağlar.

Şu anda çalışan işlemin bağlamını kaydetmek için, işletim sistemi, şu anda çalışan işlemin genel amaçlı düzenleyicilerini, PC'sini ve çekirdek yığını işaretçisini kaydetmek için bazı düşük seviyeli derleme kodlarını çıkaracak ve ardından söz konusu kayıtları, PC'yi geri yükleyecek ve yakında yürütülecek işlem için çekirdek yığınına geçecektir. Yığınları değiştirerek, çekirdek anahtar koduna yapılan çağrıyı bir işlem bağlamında (dehşete düşmüş olan) girer ve başka bir işlem bağlamında (yakında yürütülecek olan) geri döner. İşletim sistemi sonunda bir tuzaktan dönüş talimatı yürüttüğünde,

İPUCU: KONTROLÜ YENIDEN KAZANMAK IÇIN ZAMANLAYICI KESME'YI KULLANIN

Bir **zamanlayıcı kesintisinin(timer interrupt)** eklenmesi, işlemler işbirlikçi olmayan bir şekilde hareket etse bile işletim sistemine bir CPU üzerinde tekrar çalışma yeteneği verir. Bu nedenle, bu donanım özelliği, işletim sisteminin makinenin kontrolünü korumasına yardımcı olmak için gereklidir.

İPUCU: YENIDEN BAŞLATMA YARARLIDIR

Daha önce, işbirlikçi önlem altında sonsuz döngülere (ve benzer davranışlara) tek çözümün makineyi yeniden başlatmak olduğunu belirtmiştik. Bu hack'te alay edebilirken, araştırmacılar yeniden başlatmanın (veya genel olarak, bir yazılım parçasından başlayarak) sağlam sistemler oluşturmada oldukça yararlı bir araç olabileceğini göstermiştir [C + 04].

Özellikle, yeniden başlatma yararlıdır, çünkü yazılımı bilinen ve muhtemelen daha fazla test edilmiş bir duruma geri taşır. Yeniden başlatmalar ayrıca, aksi takdirde kullanımı zor olabilecek eski veya sızdırılmış yeniden kaynakları (örneğin, bellek) geri kazanır. Son olarak, yeniden başlatmaların otomatikleştirilmesi kolaydır. Tüm bu nedenlerden dolayı, sistem yönetimi yazılımı için büyük ölçekli küme Internet hizmetlerinde, makine kümelerini sıfırlamak ve böylece yukarıda listelenen avantajları elde etmek için düzenli aralıklarla yeniden başlatılması nadir değildir.

Böylece, bir dahaki sefere yeniden başlattığınızda, sadece çirkin bir hack uygulamıyorsunuz. Bunun yerine, bir bilgisayar sisteminin davranışını geliştirmek için zaman içinde sınanmış bir yaklaşım kullanıyorsunuz. Bravo!

yakında yürütülecek olan işlem, şu anda çalışan işlem haline gelir. Ve böylece bağlam anahtarı tamamlanmış olur.

Tüm sürecin zaman çizelgesi Şekil 6.3'te gösterilmiştir. Bu örnekte, İşlem A çalışıyor ve sonra zamanlayıcı kesme tarafından kesintiye uğruyor. Donanım kayıtlarını (çekirdek yığınına) kaydeder ve çekirdeğe girer (çekirdek moduna geçer). Zamanlayıcı kesme işleyicisinde, işletim sistemi A İşlemini çalıştırmaktan B İşlemine geçmeye karar verir. Bu noktada, geçerli kayıt değerlerini (A'nın işlem yapısına) dikkatlice kaydeden, İşlem B'nin kayıtlarını (işlem yapısı girişinden) geri yükleyen ve ardından özellikle yığın işaretçisini B'nin çekirdek yığınını (A'nın değil) kullanacak şekilde değiştirerek **bağlamları değiştiren(switches context)** switch() yordamını çağırır. Son olarak, işletim sistemi, B'nin kayıtlarını geri yükleyen ve çalıştırmaya başlayan tuzaktan döner.

Bu protokol sırasında gerçekleşen iki tür kayıt kaydı/geri yükleme olduğunu unutmayın. Birincisi, zamanlayıcı kesintisinin gerçekleştiği zamandır; bu durumda, çalışan işlemin kullanıcı kayıtları, bu işlemin çekirdek yığını kullanılarak donanım tarafından örtülü olarak kaydedilir. İkincisi, işletim sisteminin A'dan B'ye geçmeye karar vermesidir; bu durumda, çekirdek yazmaçları yazılım (yani işletim sistemi) tarafından pratik olarak kaydedilir, ancak bu sefer işlemin işlem yapısında belleğe kaydedilir. İkinci eylem, sistemi A'dan çekirdeğe hapsolmuş gibi çalışmaktan B'den çekirdeğe hapsolmuş gibi hareket ettirir.

Böyle bir anahtarın nasıl yürürlüğe girdiğine dair daha iyi bir fikir vermek için, Şekil 6.4 xv6 için bağlam anahtarı kodunu göstermektedir. Bunu anlamlandırıp anlamlandıramayacağınıza bakın (bunu yapmak için biraz x86'nın yanı sıra bazı xv6'ları da bilmeniz gerekir). Eski ve yeni bağlam yapıları sırasıyla eski ve yeni sürecin süreç yapılarında bulunur.

OS @ boot (kernel mode)	Hardware
tuzak tablosunu başlat	
	adresleri hatırlayın... syscall işleyicisi zamanlayıcı işleyicisi
kesme zamanlayıcısını başlat	başlangıç zamanlayıcısı CPU'yu X ms cinsinden kesme

OS @ run (kernel mode)	Hardware	Program (user mode)
		Süreç A ...
	zamanlayıcı kesme regs(A) kaydet → k-stack(A) çekirdek moduna taşı tuzak işleyicisine atla	
Tuzağı ele alın Çağrı anahtarı() yordamı kaydetme regs(A) → proc t(A) geri yükleme regs(B) ← proc t(B) anahtarı k-stack(B)'ye tuzaktan dönüş (B'ye)	regs(B)'yi geri yükleme ← k-stack(B) kullanıcı moduna geçme B'nin bilgisayarına atla	
		Süreç B ...

Şekil 6.3: **Sınırlı Doğrudan Yürütme Protokolü (Zamanlayıcı Kesme)**

6.4 Eşzamanlılık konusunda endişeli misiniz?

Bazılarınız, dikkatli ve düşünceli okuyucular olarak, şimdi şöyle düşünüyor olabilirsiniz: "Hmm... bir sistem çağrısı sırasında bir zamanlayıcı kesintisi meydana geldiğinde ne olur?" veya "Bir kesintiyi ele alırken diğeri olduğunda ne olur? Bunun çekirdekte ele alınması zorlaşmıyor mu?" İyi sorular - henüz sizin için gerçekten biraz umudumuz var!

Cevap evet, işletim sisteminin gerçekten de kesme veya tuzak işleme sırasında başka bir kesinti meydana gelirse ne olacağı konusunda endişelenmesi gerekiyor. Aslında bu, bu kitabın ikinci parçasının tamamının tam konusu, **eşzamanlılık(concurrency)** üzerine; ayrıntılı bir tartışmayı o zamana kadar erteleyeceğiz.

İştahınızı kabartmak için, işletim sisteminin bu zor durumları nasıl ele aldığına dair bazı temel bilgileri çizeceğiz. Bir işletim sisteminin yapabileceği basit bir şey, kesme işlemi sırasında **kesilebilir kesintilerdir(disable interrupts)**; bunu yapmak,

```
1  # void swtch(struct context **old, struct context *new);
2  #
3  # Save current register context in old
4  # and then load register context from new.
5  .globl swtch
6  swtch:
7    # Save old registers
8    movl 4(%esp), %eax  # put old ptr into eax
9    popl 0(%eax)        # save the old IP
10   movl %esp, 4(%eax)  # and stack
11   movl %ebx, 8(%eax)  # and other registers
12   movl %ecx, 12(%eax)
13   movl %edx, 16(%eax)
14   movl %esi, 20(%eax)
15   movl %edi, 24(%eax)
16   movl %ebp, 28(%eax)
17
18   # Load new registers
19   movl 4(%esp), %eax  # put new ptr into eax
20   movl 28(%eax), %ebp # restore other registers
21   movl 24(%eax), %edi
22   movl 20(%eax), %esi
23   movl 16(%eax), %edx
24   movl 12(%eax), %ecx
25   movl 8(%eax), %ebx
26   movl 4(%eax), %esp  # stack is switched here
27   pushl 0(%eax)       # return addr put in place
28   ret                 # finally return into new ctxt
```

Şekil 6.4: xv6 Bağlam Anahtarı Kodu

bir kesinti işleniyor, başka hiç kimse CPU'ya teslim edilmeyecek. Tabii ki, işletim sistemi bunu yaparken dikkatli olmalı; Kesintileri çok uzun süre devre dışı bırakmak, (teknik açıdan) kötü olan kayıp kesintilere yol açabilir.

İşletim sistemleri ayrıca dahili veri yapılarına eşzamanlı erişimi korumak için bir dizi gelişmiş **kilitleme(locking)** şeması geliştirmiştir. Bu, çekirdek içinde birden fazla etkinliğin aynı anda devam etmesini sağlar, özellikle de çok işlemcili işlemcilerde kullanışlıdır. Bununla birlikte, eşzamanlılık hakkındaki bu kitabın bir sonraki bölümünde göreceğimiz gibi, bu tür kilitleme birleştirilebilir ve çeşitli ilginç ve bulunması zor hatalara yol açabilir.

6.4 Özet

Toplu olarak sınırlı doğrudan yürütme olarak adlandırdığımız bir dizi teknik olan CPU sanallaştırmayı uygulamak için bazı önemli **düşük seviyeli mekanizmaları(Limited direct execution)** tanımladık. Temel fikir basittir: sadece CPU'da çalıştırmak istediğiniz programı çalıştırın, ancak önce işlemin işletim sistemi yardımı olmadan neler yapabileceğini sınırlamak için donanımı kurduğunuzdan emin olun..

BIR KENARA: BAĞLAM ANAHTARLARI NE KADAR SÜRER?

Aklınıza gelebilecek doğal bir soru şudur: bağlam değişikliği gibi bir şey ne kadar sürer? Ya da hatta bir sistem çağrısı? Sevimli olanlarınız için, tam olarak bu şeyleri ölçen **lmbench** [MS96] adlı bir araç ve aynı zamanda göreceli olabilecek birkaç performans ölçüsü daha vardır.

Sonuçlar zaman içinde biraz iyileşti ve kabaca işlemci performansını izledi. Örneğin, 1996'da Linux 1.3.37'yi 200 MHz P6 CPU'da çalıştırırken, sistem çağrıları yaklaşık 4 mikrosaniye sürdü ve bir bağlam anahtarı yaklaşık 6 mikrosaniye sürdü [MS96]. Modern sistemler, 2 veya 3 GHz işlemcili sistemlerde mikrosaniyenin altında sonuçlarla neredeyse bir miktar daha iyi performans gösterir.

Tüm işletim sistemi eylemlerinin CPU performansını izlemediğine dikkat edilmelidir. Ousterhout'un gözlemlediği gibi, birçok işletim sistemi işlemi bellek yoğundur ve bellek bant genişliği zaman içinde işlemci hızı kadar önemli ölçüde gelişmemiştir [O90]. Bu nedenle, iş yükünüze bağlı olarak, en yeni ve en iyi işlemciyi satın almak, işletim sisteminizi umduğunuz kadar hızlandırmayabilir.

Bu genel yaklaşım gerçek hayatta da benimsenir. Örneğin, çocuk sahibi olanlarınız veya en azından çocukları duymuş olanlarınız, bir odanın bebek geçirmezliği kavramına aşina olabilirsiniz: tehlikeli şeyleri tutan ve elektrik prizlerini kaplayan dolapları kilitlemek. Oda bu şekilde hazırlandığında, bebeğinizin serbestçe dolaşmasına izin verebilir, odanın en tehlikeli yönlerinin kısıtlandığını bilerek güvence altına alabilirsiniz.

Benzer şekilde, işletim sistemi, önce (önyükleme süresi sırasında) tuzak işleyicilerini ayarlayarak ve bir kesme zamanlayıcısı başlatarak ve ardından işlemleri yalnızca sınırlı bir modda çalıştırarak CPU'yu "bebek kanıtı" yapar. Bunu yaparak, işletim sistemi, işlemlerin verimli bir şekilde çalışabileceğinden, yalnızca ayrıcalıklı işlemleri gerçekleştirmek için işletim sistemi müdahalesi gerektirdiğinden veya CPU'yu çok uzun süre tekelleştirdiklerinde ve bu nedenle kapatılması gerektiğinde oldukça emin olabilir.

Böylece CPU'yu sanallaştırmak için temel mekanizmalara sahibiz. Ancak önemli bir soru cevapsız bırakılıyor: Belirli bir zamanda hangi süreci yürütmeliyiz? Zamanlayıcının cevaplaması gereken bu soru ve dolayısıyla çalışmamızın bir sonraki konusu.

BIR KENARA: ANAHTAR CPU SANALLAŞTIRMA TERİMLERİ (MEKANİZMALAR)

6.4.1 CPU en az iki yürütme modunu desteklemelidir: kısıtlanmış **kullanıcı modu(user mode)** ve ayrıcalıklı (kısıtlanmamış) **çekirdek modu(kernel mode).**

6.4.2 Tipik kullanıcı uygulamaları kullanıcı modunda çalışır ve bir **sistem çağrısı(system call)** kullanır
işletim sistemi hizmetlerini istemek üzere çekirdeğe **hapsetmek(trap)** için.

6.4.3 Tuzak yönergesi, kayıt durumunu dikkatli bir şekilde kaydeder, sabit yazılım durumunu çekirdek modunu değiştirir ve işletim sistemine önceden belirlenmiş bir hedefe atlar: **tuzak tablosu(trap table)**.

6.4.4 İşletim sistemi bir sistem çağrısına hizmet vermeyi bitirdiğinde, ayrıcalığı azaltan ve işletim sistemine atlayan **tuzaktan sonra kontrolü talimata döndüren(return-from trap**) başka bir özel tuzaktan dönüş talimatı aracılığıyla kullanıcı programına geri döner.

6.4.5 Tuzak tabloları önyükleme sırasında işletim sistemi tarafından kurulmalı ve kullanıcı programları tarafından kolayca değiştirilemediklerinden emin olunmalıdır. Tüm bunlar, programları verimli bir şekilde çalıştıran, ancak işletim sistemi kontrolünü kaybetmeden **sınırlı doğrudan yürütme(limited direct execution)** protokolünün bir parçasıdır.

6.4.6 Bir program çalışmaya başladıktan sonra, işletim sistemi, kullanıcı programının sonsuza dek çalışmamasını sağlamak için donanım mekanizmalarını, yani **zamanlayıcı kesintisini(timer interrupt)** kullanmalıdır. Bu yaklaşım, CPU zamanlaması için **işbirlikçi olmayan(non-cooperative)** bir yaklaşımdır.

6.4.7 Bazen işletim sistemi, bir zamanlayıcı kesintisi veya sistem çağrısı sırasında, geçerli işlemi çalıştırmaktan farklı bir işleme, **bağlam anahtarı(context switch)** olarak bilinen düşük seviyeli bir tekniğe geçmek isteyebilir.

.

KAYNAKÇA

[A79] "Alto User's Handbook" by Xerox. Xerox Palo Alto Research Center, September 1979. Available: http://history-computer.com/Library/AltoUsersHandbook.pdf. *An amazing system, way ahead of its time. Became famous because Steve Jobs visited, took notes, and built Lisa and eventually Mac.*

[C+04] "Microreboot — A Technique for Cheap Recovery" by G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, A. Fox. OSDI '04, San Francisco, CA, December 2004. *An excellent paper pointing out how far one can go with reboot in building more robust systems.*

[I11] "Intel 64 and IA-32 Architectures Software Developer's Manual" by Volume 3A and 3B: System Programming Guide. Intel Corporation, January 2011. *This is just a boring manual, but sometimes those are useful.*

[K+61] "One-Level Storage System" by T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner. IRE Transactions on Electronic Computers, April 1962. *The Atlas pioneered much of what you see in modern systems. However, this paper is not the best one to read. If you were to only read one, you might try the historical perspective below [L78].*

[L78] "The Manchester Mark I and Atlas: A Historical Perspective" by S. H. Lavington. Communications of the ACM, 21:1, January 1978. *A history of the early development of computers and the pioneering efforts of Atlas.*

[M+63] "A Time-Sharing Debugging System for a Small Computer" by J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider. AFIPS '63 (Spring), May, 1963, New York, USA. *An early paper about time-sharing that refers to using a timer interrupt; the quote that discusses it: "The basic task of the channel 17 clock routine is to decide whether to remove the current user from core and if so to decide which user program to swap in as he goes out."*

[MS96] "lmbench: Portable tools for performance analysis" by Larry McVoy and Carl Staelin. USENIX Annual Technical Conference, January 1996. *A fun paper about how to measure a number of different things about your OS and its performance. Download lmbench and give it a try.*

[M11] "Mac OS 9" by Apple Computer, Inc.. January 2011. http://en.wikipedia.org/wiki/Mac_OS_9 . *You can probably even find an OS 9 emulator out there if you want to; check it out, it's a fun little Mac!*

[O90] "Why Aren't Operating Systems Getting Faster as Fast as Hardware?" by J. Ousterhout. USENIX Summer Conference, June 1990. *A classic paper on the nature of operating system performance.*

[P10] "The Single UNIX Specification, Version 3" by The Open Group, May 2010. Available: http://www.unix.org/version3/. *This is hard and painful to read, so probably avoid it if you can. Like, unless someone is paying you to read it. Or, you're just so curious you can't help it!*

[S07] "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" by Hovav Shacham. CCS '07, October 2007. *One of those awesome, mind-blowing ideas that you'll see in research from time to time. The author shows that if you can jump into code arbitrarily, you can essentially stitch together any code sequence you like (given a large code base); read the paper for the details. The technique makes it even harder to defend against malicious attacks, alas.*

Ödev (Ölçüm)

BIR KENARA: ÖLÇÜM ÖDEVLERI

Ölçüm ödevleri, işletim sistemi veya donanım performansının bazı yönlerini ölçmek için gerçek bir makinede çalıştırmak üzere kod yazdığınız küçük alıştırmalardır. Bu tür ev ödevlerinin arkasındaki fikir, size gerçek bir işletim sistemi ile biraz uygulamalı deneyim kazandırmaktır.

Bu ödevde, bir sistem çağrısının ve bağlam anahtarının maliyetlerini ölçeceksiniz. Bir sistem çağrısının maliyetini ölçmek nispeten kolaydır. Örneğin, basit bir sistem çağrısını tekrar tekrar çağırabilir (örneğin, 0 baytlık bir okuma gerçekleştirerek) ve bunun ne kadar süreceğini sağlayabilirsiniz; Süreyi yineleme sayısına bölmek, bir sistem çağrısının maliyetine ilişkin bir tahmin verir.

Dikkate almanız gereken bir şey, zamanlayıcınızın hassasiyeti ve doğruluğudur. Kullanabileceğiniz tipik bir zamanlayıcı gettimeofday(); ayrıntılar için ana sayfasını okuyun. Orada göreceğiniz şey, gettimeofday() öğesinin 1970'ten bu yana zamanı mikrosaniye cinsinden döndürdüğü; ancak bu, zamanlayıcının mikrosaniye için hassas olduğu anlamına gelmez. Arka arkaya aramaları ölçme

gettimeofday() için zamanlayıcının ne kadar hassas olduğu hakkında bir şeyler öğrenmek için; bu, iyi bir ölçüm sonucu elde etmek için boş sistem çağrısı testinizin kaç yinelemesini çalıştırmanız gerektiğini size söyleyecektir. gettimeofday() sizin için yeterince kesin değilse, x86 makinelerinde bulunan rdtsc yönergesini kullanmayı düşünebilirsiniz.

Bir bağlam anahtarının maliyetini ölçmek biraz daha zordur. lmbench kıyaslaması bunu tek bir CPU üzerinde iki işlem çalıştırarak ve aralarında iki UNIX kanalı ayarlayarak yapar; Boru, UNIX sistemindeki işlemlerin birbirleriyle iletişim kurabilmesinin birçok yolundan yalnızca biridir. İlk işlem daha sonra ilk boruya bir yazma yayınlar ve ikincisinde bir okuma bekler; İkinci borudan bir şeylerin okunmasını bekleyen ilk işlemi gördükten sonra, işletim sistemi ilk işlemi bloke durumuna geçirir ve ilk borudan okuyan ve ardından ikinciye yazan diğer işleme geçer. İkinci işlem ilk borudan tekrar okumaya çalıştığında, tıkanır ve böylece iletişimin ileri geri döngüsü devam eder. lmbench, bu şekilde iletişim kurmanın maliyetini tekrar tekrar ölçerek, bir bağlam anahtarının maliyetini iyi bir şekilde tahmin edebilir. Burada da benzer bir şeyi, boruları veya UNIX yuvaları gibi başka bir iletişim mekanizmasını kullanarak yeniden oluşturmayı deneyebilirsiniz.

Bağlam değiştirme maliyetini ölçmede bir zorluk, birden fazla CPU'ya sahip sistemlerde ortaya çıkar; Böyle bir sistemde yapmanız gereken, bağlam değiştirme süreçlerinizin aynı işlemci üzerinde bulunduğundan emin olmaktır. Ne yazık ki, çoğu işletim sisteminin bir işlemi kısmi bir işlemciye bağlama çağrıları vardır; Linux'ta, örneğin, sched setaffinity() çağrısı aradığınız şeydir. Her iki işlemin de aynı işlemcide olmasını sağlayarak, işletim sisteminin bir işlemi durdurup diğerini aynı CPU'ya geri yüklemesinin maliyetini ölçtüğünüzden emin olursunuz.

```
#include <time.h>
#include <unistd.h>

// Örnek bir sistem çağrısı (0-baytlık bir okuma)
int null_syscall() {
   char buf[1];
   return read(0, buf, 0);
}

int main() {
   // Ölçümleri yapmak için gerekli değişkenler
   struct timeval start, end;
   long elapsed_time;
   int num_iterations = 1000; // Ölçümlerin kaç kez tekrar edeceği

   // Zamanlayıcıyı başlat
   gettimeofday(&start, NULL);

   // Örnek sistem çağrısını tekrar tekrar çalıştır
   for (int i = 0; i < num_iterations; i++) {
      null_syscall();
   }

   // Zamanlayıcıyı durdur ve geçen zamanı hesapla
   gettimeofday(&end, NULL);
   elapsed_time = (end.tv_sec - start.tv_sec) * 1000000;
   elapsed_time += end.tv_usec - start.tv_usec;

   // Bir sistem çağrısının maliyetini hesapla
   double cost_per_syscall = (double) elapsed_time / num_iterations;

   // Sonuçları ekrana yazdır
   printf("Elapsed time: %ld microseconds\n", elapsed_time);
   printf("Cost per syscall: %f microseconds\n", cost_per_syscall);

   return 0;
}
```