Mekanizma: Sınırlı Doğrudan Yürütme

CPU'yu sanallaştırmak için, işletim sisteminin fiziksel CPU'yu görünüşte aynı anda çalışan birçok iş arasında bir şekilde paylaşması gerekir. Temel fikir basittir: bir işlemi kısa bir süre çalıştırın, ardından başka bir işlemi çalıştırın ve bu şekilde devam edin. CPU'nun bu şekilde paylaşılmasıyla (**time sharing)** sanallaştırma sağlanır.
Bununla birlikte, bu tür sanallaştırma makinelerini oluşturmanın birkaç zorluğu vardır. Birincisi performans: Sisteme aşırı yük eklemeden sanallaştırmayı nasıl uygulayabiliriz? İkincisi kontrol: İşlemci üzerinde kontrolü sürdürürken süreçleri verimli bir şekilde nasıl çalıştırabiliriz? Kontrol, kaynaklardan sorumlu olduğu için işletim sistemi için özellikle önemlidir; kontrol olmadan, bir süreç basitçe sonsuza kadar çalışabilir ve makineyi ele geçirebilir veya erişmesine izin verilmemesi gereken bilgilere erişebilir. Bu nedenle, kontrolü sürdürürken yüksek performans elde etmek, bir işletim sistemi oluşturmanın temel zorluklarından biridir.

Önemli Nokta:
İŞLEMCİ KONTROL İLE VERİMLİ BİR ŞEKİLDE SANALLAŞTIRILIR
İşletim sistemi, sistem üzerinde kontrolü elinde tutarken CPU'yu verimli bir şekilde sanallaştırmalıdır. Bunu yapmak için hem donanım hem de işletim sistemi desteği gerekli olacaktır. İşletim sistemi, işini etkili bir şekilde gerçekleştirmek için genellikle mantıklı bir donanım desteği kullanır.

6.1 Temel Teknik: Sınırlı Doğrudan Uygulama
Bir programın beklendiği kadar hızlı çalışmasını sağlamak için, işletim sistemi geliştiricilerinin sınırlı doğrudan yürütme (**limited direct execution)** dediğimiz bir teknik bulmaları şaşırtıcı değildir. Fikrin "doğrudan yürütme" kısmı basittir: programı doğrudan CPU üzerinde çalıştırmanız yeterlidir.
Böylece, işletim sistemi bir programı çalıştırmak istediğinde, bir işlem listesinde onun için bir işlem girişi oluşturur, bunun için bir miktar bellek ayırır, program kodunu belleğe (diskten) yükler, giriş noktasını bulur (yani, main() rutini veya benzeri bir şey), atlar

İşletim Sistemin (**OS**)	Program
İşlem listesi için giriş oluştur	
Program için bellek tahsis et	
Programı belleğe yükle	
argc/argv ile yığın oluştur	
Kayıtları temizle main() çağrısını (**call**) yürüt	

main()'i çalıştır
main'den dönüşü (**return**) yürüt

İşlemin boş hafızasını

İşlemtim listesinden kaldır

Şekil 6.1: Doğrudan Yürütme Protokolü (Sınırsız)

(Direct Execution Protocol (Without Limits))

ve kullanıcının kodunu çalıştırmaya başlar. Şekil 6.1, programın ana işlevine () ve daha sonra çekirdeğe geri atlamak için normal bir arama ve dönüş kullanan bu temel doğrudan yürütme protokolünü (henüz herhangi bir sınırlama olmaksızın) göstermektedir.
Kulağa basit geliyor, değil mi? Ancak bu yaklaşım, CPU'yu sanallaştırma arayışımızda birkaç soruna yol açıyor. İlki basit: Eğer sadece bir program çalıştırırsak, işletim sistemi, programı verimli bir şekilde çalıştırırken, programın yapmasını istemediğimiz hiçbir şeyi yapmadığından nasıl emin olabilir? İkincisi: Bir işlemi çalıştırdığımızda, işletim sistemi nasıl onun çalışmasını durdurur ve başka bir işleme geçer, böylece CPU'yu sanallaştırmak için ihtiyaç duyduğumuz zaman paylaşımını (**time sharing**) gerçekleştirir?
Aşağıda bu soruları yanıtlarken, CPU'yu sanallaştırmak için neyin gerekli olduğunu çok daha iyi anlayacağız. Bu teknikleri geliştirirken, ismin “sınırlı” kısmının nereden geldiğini de göreceğiz; çalışan programlarda sınırlamalar olmadan, işletim sistemi hiçbir şeyi kontrol edemez ve bu nedenle "sadece bir kitaplık" olur - gelecek vadeden bir işletim sistemi için çok üzücü bir durum!

6.2 Problem 1: Kısıtlanmış İşlemler

Doğrudan yürütme, hızlı olmanın bariz avantajına sahiptir; program yerel olarak donanım CPU'sunda çalışır ve bu nedenle beklendiği kadar hızlı yürütülür. Ancak CPU üzerinde çalışmak bir sorun ortaya çıkarır: İşlem, bir diske G/Ç isteği göndermek veya CPU veya bellek gibi daha fazla sistem kaynağına erişim elde etmek gibi bir tür kısıtlı işlem gerçekleştirmek isterse ne olur?

ÖNEMLİ NOKTA: KISITLANMIŞ İŞLEMLER NASIL GERÇEKLEŞTİRİLİR?

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

KENARA: SİSTEM ÇAĞRILARI NEDEN PROSEDÜR ÇAĞRILARI GİBİ GÖRÜNÜYOR?

open() veya read() gibi bir sistem çağrısı çağrısının neden C'deki tipik bir prosedür çağrısı gibi göründüğünü merak edebilirsiniz; yani, tıpkı bir prosedür

çağrısı gibi görünüyorsa, sistem bunun bir sistem çağrısı olduğunu nasıl biliyor ve tüm doğru şeyleri yapıyor? Basit sebep: Bu bir prosedür çağrısıdır, ancak bu prosedür çağrısının içinde ünlü tuzak talimatı gizlidir. Daha spesifik olarak, open() işlevini çağırdığınızda (örneğin), C kitaplığına bir prosedür çağrısı yürütüyorsunuz. Burada, open() veya sağlanan diğer sistem çağrılarından herhangi biri için, kitaplık, argümanları iyi bilinen konumlara (örn. veya belirli kayıtlarda), sistem çağrısı numarasını da iyi bilinen bir konuma koyar (yine yığına veya bir kayda) ve ardından yukarıda bahsedilen tuzak talimatını yürütür. Tuzak paketini açtıktan sonra kitaplıktaki kod, dönüş değerlerini açar ve kontrolü sistem çağrısını yapan programa geri verir. Bu nedenle, C kitaplığının sistem çağrıları yapan bölümleri, donanıma özgü tuzak talimatını yürütmenin yanı sıra argümanları işlemek ve değerleri doğru bir şekilde döndürmek için kurallara dikkatle uymaları gerektiğinden, derlemede elle kodlanmıştır. Ve artık bir işletim sistemine tuzak kurmak için kişisel olarak neden derleme kodu yazmak zorunda olmadığınızı biliyorsunuz; birisi o derlemeyi sizin için zaten yazdı.

Bir yaklaşım, herhangi bir sürecin I/O ve diğer ilgili işlemler açısından istediğini yapmasına izin vermek olacaktır. Bununla birlikte, bunu yapmak, arzu edilen birçok sistem türünün inşasını engelleyecektir. Örneğin, bir dosyaya erişim izni vermeden önce izinleri denetleyen bir dosya sistemi oluşturmak istiyorsak, herhangi bir kullanıcının diske I/O düzenlemesine izin veremeyiz; bunu yaparsak, bir işlem tüm diski okuyabilir veya yazabilir ve bu nedenle tüm korumalar kaybolur.
Bu nedenle, benimsediğimiz yaklaşım, kullanıcı modu (**user mode)** olarak bilinen yeni bir işlemci modunu tanıtmaktır; kullanıcı modunda çalışan kodun yapabilecekleri sınırlıdır. Örneğin, kullanıcı modunda çalışırken, bir işlem I/O isteklerini yayınlayamaz; bunu yapmak, işlemcinin bir istisna oluşturmasına neden olur; işletim sistemi daha sonra muhtemelen süreci öldürür.
Kullanıcı modunun aksine, işletim sisteminin (veya çekirdeğin) içinde çalıştığı çekirdek modu **(kernel mode)** vardır. Bu modda çalışan kod, I/O isteklerini yayınlama ve her türlü kısıtlı talimatlar
Ancak yine de bir zorlukla karşı karşıyayız: Bir kullanıcı işlemi, diskten okuma gibi bir tür ayrıcalıklı işlem gerçekleştirmek istediğinde ne yapmalıdır? Bunu sağlamak için, hemen hemen tüm modern donanımlar, kullanıcı programlarının bir sistem çağrısı gerçekleştirmesini sağlar. Atlas [K+61,L78] gibi eski makinelere öncülük eden sistem çağrıları(**system call)**, çekirdeğin, dosya sistemine erişim, süreçler oluşturma ve yok etme, diğerleriyle iletişim kurma gibi bazı önemli işlevsellik parçalarını kullanıcı programlarına dikkatlice göstermesine izin verir. süreçler ve daha fazlasını hafızaya tahsis eder.

İPUCU: KORUMALI KONTROL AKTARIMI KULLANIN

Donanım, farklı yürütme modları sağlayarak işletim sistemine yardımcı olur. Kullanıcı modunda **(user mode),** uygulamaların donanım kaynaklarına tam erişimi yoktur. Çekirdek modunda **(kernel mode)**, işletim sisteminin makinenin

tüm kaynaklarına erişimi vardır. Çekirdeğe tuzak kurmak**(return-from-trap)** ve tuzaktan **(trap)** kullanıcı modu programlarına geri dönmek için özel talimatlar ve işletim sisteminin donanıma tuzak tablosunun **(trap table)** bellekte nerede olduğunu söylemesine izin veren talimatlar da sağlanır.

Çoğu işletim sistemi birkaç yüz arama sağlar (ayrıntılar için POSIX standardına bakın [P10]); erken Unix sistemleri, yaklaşık yirmi çağrıdan oluşan daha özlü bir alt küme ortaya çıkardı.
Bir sistem çağrısını yürütmek için, bir programın özel bir tuzak **(trap)** komutunu yürütmesi gerekir. Bu talimat aynı anda çekirdeğe atlar ve ayrıcalık seviyesini çekirdek moduna yükseltir; çekirdeğe girdikten sonra, sistem artık gereken ayrıcalıklı işlemleri gerçekleştirebilir (eğer izin verilirse) ve böylece çağırma işlemi için gerekli işi yapabilir. Bittiğinde, işletim sistemi özel bir tuzaktan dönüş talimatını çağırır**(return-from-trap)** ve bu, beklediğiniz gibi, aynı anda ayrıcalık seviyesini tekrar kullanıcı moduna düşürürken çağıran kullanıcı programına geri döner.
Bir tuzak yürütülürken donanımın biraz dikkatli olması gerekir, çünkü işletim sistemi tuzaktan dönüş talimatını verdiğinde doğru şekilde geri dönebilmek için arayanın kayıtlarından yeterince tasarruf ettiğinden emin olmalıdır. Örneğin, x86'da işlemci, program sayacını, bayrakları ve diğer birkaç kaydı işlem başına bir çekirdek yığınına**(kernel stack)** gönderir; tuzaktan dönüş, bu değerleri yığından çıkaracak ve kullanıcı modu programının yürütülmesine devam edecektir (ayrıntılar için Intel sistem kılavuzlarına [I11] bakın). Diğer donanım sistemleri farklı kurallar kullanır, ancak temel kavramlar platformlar arasında benzerdir.
Bu tartışmanın dışında kalan önemli bir ayrıntı var: Tuzak işletim sisteminde hangi kodu çalıştıracağını nasıl biliyor? Açıkçası, arama süreci atlanacak bir adres belirleyemez (bir prosedür çağrısı yaparken yaptığınız gibi); bunu yapmak, programların çekirdeğe herhangi bir yere atlamasına izin verir ki bu açıkça Çok Kötü Bir Fikir**(Very Bad Idea)**'dir. Bu nedenle çekirdek, bir tuzakta hangi kodun yürütüleceğini dikkatlice kontrol etmelidir.
Çekirdek bunu önyükleme sırasında bir tuzak tablosu **(trap table)** kurarak yapar. Makine açıldığında, bunu ayrıcalıklı (çekirdek) modda yapar ve böylece makine donanımını gerektiği gibi yapılandırmakta serbesttir. Bu nedenle işletim sisteminin yaptığı ilk şeylerden biri, donanıma belirli istisnai olaylar meydana geldiğinde hangi kodu çalıştıracağını söylemektir. Örneğin, bir sabit disk kesintisi gerçekleştiğinde, bir klavye kesintisi meydana geldiğinde veya bir program bir sistem çağrısı yaptığında hangi kod çalıştırılmalıdır? İşletim sistemi donanım hakkında bilgi verir.

İşletim Sistemine @ önyükleme Donanım

(çekirdek modu)

---

tuzak tablosunu başlat**(initialize trap table)**

adresini hatırla...

sistem çağrısı işleyicisi

İşletim Sistemini @ çalıştır Donanım Program

(çekirdek modu) (kullanıcı modu)

İşlem listesi için girdi oluştur

Program için bellek ayır

Programı belleğe yükle

Kurulum argv Dosyası ile

kullanıcı yığını reg / PC ile

çekirdek yığını tuzaktan dönüştür**(return-from-trap)**

reg'leri geri yükle

(çekirdek yığınından)

kullanıcı moduna geç

ana moda atla

main()'i çalıştır

...

Çağrı sistemi çağrısı

işletim sistemine tuzak**(trap)**

kayıtları kaydet
(çekirdek yığınına)

çekirdek moduna geç

tuzak işleyicisine atla

Tutma tuzağı
Sistem çağrısı işini yap
tuzaktan dönüş**(return-from-trap)**

kayıtları geri yükle
(çekirdek yığınından)

kullanıcı moduna geç

tuzaktan sonra PC'ye atla

...

main'den dönüş

tuzak**(trap)** (çıkış yoluyla ())

Boş işlem belleği

İşlem listesinden kaldır

Şekil 6.2: Sınırlı Doğrudan Yürütme Protokolü

(Limited Direct Execution Protocol)

Bu tuzak işleyicilerinin **(trap handlers)** yerleri, genellikle bir tür özel talimatla. Donanım bilgilendirildikten sonra, makine bir sonraki yeniden başlatılıncaya kadar bu işleyicilerin konumunu hatırlar ve böylece donanım, sistem çağrıları ve diğer istisnai olaylar gerçekleştiğinde ne yapacağını (yani hangi koda geçileceğini) bilir.

İPUCU: GÜVENLİ SİSTEMLERDEKİ KULLANICI GİRDİLERİNE KARŞI DİKKATLİ OLUN

Sistem çağrıları sırasında işletim sistemini korumak için büyük çaba sarf etmemize rağmen (bir donanım yakalama mekanizması ekleyerek ve işletim sistemine yapılan tüm çağrıların bu mekanizma üzerinden yönlendirilmesini sağlayarak), güvenli **(secure)** bir işletim sistemini basitleştirmenin dikkate almamız gereken birçok başka yönü daha var. Bunlardan biri, sistem çağrısı sınırındaki bağımsız değişkenlerin işlenmesidir; İşletim sistemi, kullanıcının ne geçtiğini kontrol etmeli ve bağımsız değişkenlerin doğru şekilde belirtildiğinden emin olmalı veya çağrıyı başka bir şekilde reddetmelidir.

Örneğin, bir write() sistem çağrısı ile kullanıcı, yazma çağrısının kaynağı olarak arabelleğin adresini belirtir. Kullanıcı (yanlışlıkla veya kötü niyetle) “kötü” bir adreste (örneğin, çekirdeğin adres alanının içindeki bir adres) geçerse, işletim sistemi bunu algılamalı ve aramayı 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 sağlayacaktır.

Genel olarak, güvenli bir sistem kullanıcı girdilerini büyük bir şüpheyle ele almalıdır. Bunu yapmamak şüphesiz kolayca saldırıya uğramış yazılımlara, dünyanın güvensiz ve korkutucu bir yer olduğuna dair umutsuzluğa ve çok güvenen işletim sistemi geliştiricisi için iş güvenliği kaybına yol açacaktır.

Tam sistem çağrısını belirtmek için, bir sistem çağrısı numarası **(system-call number)** genellikle her sistem çağrısı için aynı şekilde imzalanır. Bu nedenle kullanıcı kodu, istenen sistem çağrısı numarasını bir kayıt defterine veya yığındaki belirli bir konuma yerleştirmekten sorumludur; işletim sistemi, sistem çağrısını tuzak işleyicisinin içine işlerken bu numarayı inceler, geçerli olmasını sağlar ve eğer öyleyse, karşılık gelen kodu yürütür. Bu yönlendirme düzeyi bir koruma**(protection)** biçimi olarak hizmet eder; kullanıcı kodu, atlanacak tam bir adres belirtemez, bunun yerine numara aracılığıyla belirli bir hizmet talep etmelidir.

Son bir kenara: Tuzak tablolarının nerede olduğunu donanıma söyleme talimatını yerine getirebilmek çok güçlü bir yetenektir. Bu nedenle, tahmin edebileceğiniz gibi, aynı zamanda ayrıcalıklı **(privileged)** bir işlemdir. Bu talimatı kullanıcı modunda çalıştırmaya çalışırsanız, donanım size izin vermez ve muhtemelen ne olacağını tahmin edebilirsiniz (ipucu: hoşçakalın, rahatsız edici program). Düşünmek için işaret edin: 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 özetlemektedir. Her işlemin, çekirdeğe girip çıkarken kayıtların (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. İlkinde (önyükleme sırasında), çekirdek tuzak tablosunu başlatır ve CPU sonraki kullanım için 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 ayırma, bellek ayırma) ayarlar; Bu, CPU'yu kullanıcı moduna geçirir ve çalışmaya başlar süreç. İşlem bir sistem çağrısı vermek istediğinde, onu işleyen işletim sistemine geri hapseder ve bir kez daha tuzaktan işleme dönüşü yoluyla kontrolü geri döndürür. İşlem daha sonra işini tamamlar ve main() öğesinden geri döner.; bu genellikle programdan düzgün bir şekilde çıkacak bazı saplama kodlarına geri döner (örneğin, işletim sistemine giren exit () sistem çağrısını çağırarak). Bu noktada işletim sistemi temizlenir ve işimiz biter.

6.3 Sorun # 2: İşlemler Arasında Geçiş Yapma

Doğrudan yürütme ile ilgili bir sonraki sorun, süreçler arasında bir geçiş sağlamaktır. Süreçler arasında geçiş yapmak basit olmalı, değil mi? İşletim sistemi sadece bir işlemi durdurmaya ve başka bir işlemi başlatmaya karar vermelidir. Büyütülecek ne var ki? Ama 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: olamaz) Bu kulağa neredeyse felsefi gelse de, bu

gerçek bir sorundur: İşletim sisteminin CPU üzerinde çalışmıyorsa bir işlem yapmasının açıkça bir yolu yoktur. Böylece sorunun özüne varıyoruz.

İŞİN ÖZÜ: CPU'NUN KONTROLÜ NASIL YENİDEN KAZANILIR

İşletim sistemi, işlemler arasında geçiş yapabilmesi için cpu'nun kontrolünü nasıl yeniden kazanabilir?**(regain control)**

İşbirliğine Dayalı Bir Yaklaşım: Sistem Çağrılarını Bekleyin

(A Cooperative Approach: Wait For System Calls)

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

Bu nedenle, bu ütopik dünyada dostça bir süreç cpu'dan nasıl vazgeçer diye sorabilirsiniz. Çoğu işlem, ortaya çıktığı gibi, örneğin bir dosyayı açmak ve ardından okumak veya başka bir makineye mesaj göndermek veya yeni bir işlem oluşturmak için sistem çağrıları **(system calls)** yaparak cpu'nun kontrolünü oldukça sık işletim sistemine aktarır. Bunun gibi sistemler genellikle, diğer işlemleri çalıştırabilmesi için kontrolü işletim sistemine aktarmak dışında hiçbir şey yapmayan açık bir verim **(yield)** sistemi çağrısı içerir.

Uygulamalar ayrıca yasadışı bir şey yaptıklarında kontrolü işletim sistemine aktarır. Örneğin, bir uygulama sıfıra bölünürse veya erişememesi gereken belleğe erişmeye çalışırsa, uygulama için bir tuzak**(trap)** oluşturur.

İŞLETİM. İşletim sistemi daha sonra CPU'yu tekrar kontrol edecektir (ve muhtemelen rahatsız edici işlemi sonlandıracaktır).

Böylece, işbirliğine dayalı bir zamanlama sisteminde işletim sistemi, bir sistem çağrısının veya bir tür yasa dışı işlemin gerçekleşmesini bekleyerek cpu'nun kontrolünü yeniden kazanır. Şunu da düşünüyor olabilirsiniz: Bu pasif yaklaşım idealden daha az değil mi? Örneğin, bir işlem (kötü amaçlı veya yalnızca hatalarla dolu) sonsuz bir döngüde sona ererse ve hiçbir zaman sistem çağrısı yapmazsa ne olur? İşletim sistemi o zaman ne yapabilir?

İşbirliğine Dayalı Olmayan Bir Yaklaşım: İşletim sistemi Kontrolü Ele Alır

(A Non-Cooperative Approach: The OS Takes Control)

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

İŞİN ÖZÜ: İŞBİRLİĞİ OLMADAN KONTROL NASIL KAZANILIR

İşlemler kooperatif olmasa bile işletim sistemi cpu'nun kontrolünü nasıl kazanabilir? İşletim sistemi, haydut bir işlemin makineyi devralmamasını sağlamak için ne yapabilir?

Cevabın basit olduğu ortaya çıktı ve yıllar önce bilgisayar sistemleri inşa eden birkaç kişi tarafından keşfedildi: bir zamanlayıcı kesintisi**(timer interrupt)** [M+ 63]. Bir zamanlayıcı aygıtı, her milisaniyede bir kesme oluşturacak şekilde programlanabilir; kesme yükseltildiğinde, çalışmakta olan işlem durdurulur ve işletim sisteminde 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.

Sistem çağrılarında daha önce tartıştığımız gibi, işletim sistemi, zamanlayıcı kesintisi meydana geldiğinde hangi kodun çalıştırılacağı konusunda donanımı bilgilendirmelidir; Bu nedenle, önyükleme sırasında işletim sistemi tam olarak bunu yapar. İkincisi, önyükleme sırası sırasında da işletim sistemi, elbette ayrıcalıklı olan zamanlayıcıyı başlatmalıdır

İPUCU: UYGULAMA YANLIŞ DAVRANIŞIYLA BAŞA ÇIKMAK

İşletim sistemleri genellikle tasarım (kötülük) veya kaza (hatalar) yoluyla yapmamaları gereken bir şeyi yapmaya çalışan kötü davranış süreçleriyle uğraşmak zorundadır. Modern sistemlerde, işletim sisteminin bu tür kötü muameleyle başa çıkma şekli, suçluyu basitçe sonlandırmaktır. Bir vuruş ve dışarıdasın! Belki acımasızdır, ancak belleğe yasa dışı olarak erişmeye veya yasa dışı bir talimat yürütmeye çalıştığınızda işletim sistemi başka ne yapmalıdır?

operasyon. Zamanlayıcı başladıktan sonra, işletim sistemi kendini güvende hissedebilir, böylece kontrol sonunda ona geri döner ve böylece işletim sistemi kullanıcı programlarını çalıştırmakta özgürdür. Eşzamanlılığı daha ayrıntılı olarak anladığımızda daha sonra tartışacağımız bir şey olan zamanlayıcı da kapatılabilir (aynı zamanda ayrıcalıklı bir işlem).

Donanımın, özellikle kesme gerçekleştiğinde çalışmakta olan programın durumunu yeterince kaydetmek için bir kesme oc- curs olduğunda bazı sorumlulukları olduğunu unutmayın, böylece sonraki bir tuzaktan dönüş talimatı çalışan programı doğru şekilde sürdürebilecektir. Bu eylemler kümesi, çekirdeğe açık bir sistem çağrısı tuzağı sırasında donanımın davranışına oldukça benzer, böylece çeşitli kayıtlar kaydedilir (örneğin bir çekirdek yığınına) ve böylece tuzaktan dönüş talimatı ile kolayca geri yüklenir.

Bağlamı Kaydetme ve Geri Yükleme**(Saving and Restoring Context)**

Artık işletim sistemi, ister bir sistem çağrısı yoluyla ister daha güçlü bir zamanlayıcı kesintisi yoluyla işbirliği içinde olsun, kontrolü yeniden ele geçirdiğine göre, bir karar verilmelidir: çalışmakta olan işlemi çalıştırmaya devam edip etmeyeceğine veya farklı bir işleme geçip geçmeyeceğine. Bu karar, zamanlayıcı**(scheduler)** olarak bilinen işletim sisteminin bir parçası 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: İşletim sisteminin yapması gereken tek şey, o anda yürütülmekte olan işlem için (örneğin çekirdek yığınına) birkaç kayıt değeri kaydetmek ve yakında yürütülecek işlem için birkaçını geri yüklemektir (çekirdek yığınından). Böylece işletim sistemi, tuzaktan dönüş talimatı nihayet yürütüldüğünde,

çalışmakta olan işleme geri dönmek yerine sistemin başka bir işlemin yürütülmesine devam etmesini sağlar.

Çalışmakta olan işlemin bağlamını kaydetmek için işletim sistemi, çalışmakta olan işlemin genel amaçlı regis- terlerini, pc'sini ve çekirdek yığını işaretçisini kaydetmek için bazı düşük seviyeli derleme kodlarını çıkarır ve ardından söz konusu kayıtları, PC'yi geri yükler ve çekirdek yığınına geçer yakında yürütülecek süreç için. Yığınları değiştirerek, çekirdek, bir işlem bağlamında (terrupted olan) anahtar koduna yapılan çağrıya girer ve başka bir işlem bağlamında (yakında yürütülecek olan) geri döner. İşletim sistemi nihayet bir tuzaktan dönüş talimatı yürüttüğünde,

İPUCU: KONTROLÜ YENİDEN KAZANMAK İÇİN ZAMANLAYICI KESİNTİSİNİ KULLANIN

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

İPUCU: YENİDEN BAŞLATMA YARARLIDIR

Daha önce, kooperatif önleme altındaki sonsuz döngülere (ve benzer davranışlara) tek çözümün makineyi yeniden başlatmak**(reboot)** olduğunu belirtmiştik. Bu hackle alay etseniz de, araştırmacılar yeniden başlatmanın (veya genel olarak bir yazılım parçasından başlayarak) sağlam sistemler oluşturmada oldukça faydalı bir araç olabileceğini göstermiştir [C + 04].

Özellikle, yazılımı bilinen ve muhtemelen daha test edilmiş bir duruma geri taşıdığı için yeniden başlatma kullanışlıdır. Yeniden başlatmalar ayrıca, aksi takdirde ele alınması zor olabilecek eski veya sızdırılmış yeniden kaynakları (örneğin bellek) geri alır. Son olarak, yeniden başlatmaların otomatikleştirilmesi kolaydır. Tüm bu nedenlerden dolayı, büyük ölçekli küme internet hizmetlerinde, sistem yönetim yazılımının makine setlerini sıfırlamak ve böylece yukarıda listelenen avantajları elde etmek için periyodik olarak yeniden başlatması nadir değildir.

Böylece, bir dahaki sefere yeniden başlattığınızda, sadece çirkin bir hack yapmıyorsunuz. Bunun yerine, bir bilgisayar sisteminin davranışını iyileştirmek için zamana göre test edilmiş bir yaklaşım kullanıyorsunuz. Aferin!

yakında yürütülecek süreç, o anda çalışmakta olan süreç haline gelir. Ve böylece bağlam anahtarı tamamlandı.

Tüm sürecin bir zaman çizelgesi Şekil 6.3'te gösterilmiştir. Bu örnekte, İşlem A çalışıyor ve ardından süreölçer kesmesi tarafından kesiliyor. Donanım, kayıtlarını (çekirdek yığınına) kaydeder ve çekirdeğe girer (çekirdek moduna geçer). Zamanlayıcı kesme işleyicisinde, işletim sistemi çalışan İşlem A'dan İşlem B'ye geçmeye karar verir. Bu noktada, geçerli kayıt değerlerini (A'nın işlem yapısına) dikkatli bir şekilde kaydeden, B İşleminin kayıtlarını (işlem yapısı girişinden) geri yükleyen ve ardından bağlamları değiştiren**(switches contexts)** switch() yordamını çağırır. özellikle yığın işaretçisini B'nin çekirdek yığınını kullanacak şekilde değiştirerek (ve a değil). Son olarak, işletim sistemi, B'nin kayıtlarını geri yükleyen ve çalıştırmaya başlayan tuzaktan geri döner.

Bu protokol sırasında gerçekleşen iki tür kayıt kaydetme / geri yükleme olduğunu unutmayın. Birincisi, zamanlayıcı kesintisinin meydana geldiği 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 kayıtları yazılım tarafından (yani işletim sistemi) dolaylı olarak kaydedilir, ancak bu kez 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. Bir anlam ifade edip edemeyeceğinize bakın (bunu yapmak için biraz x86 ve biraz xv6 bilmeniz gerekir). Eski ve yeni bağlam yapıları, sırasıyla eski ve yeni sürecin süreç yapılarında bulunur.

İşletim Sistemine @ önyükleme (çekirdek modu)	Donanım	
tuzak tablosunu başlat**(initialize trap table)**		
		adreslerini hatırla...
		syscall işleyicisi
		zamanlayıcı işleyicisi
kesme zamanlayıcısını başlat**(start interrupt timer)**		
		zamanlayıcıyı başlat
		CPU'yu X ms'de kesme

İşletim Sistemini @ çalıştır (çekirdek modu)	Donanım	Program (kullanıcı modu)
		Süreç A
		...
	zamanlayıcı kesme**(timer interrupt)**	
	reg'leri kaydet (A) → k-yığını (A)	
	çekirdek moduna geç	
	tuzak işleyicisine atla	
Tuzağı tut		
Çağrı anahtarı () rutin		
reg'leri kaydet (A) → proc t (A)		
reg'leri geri yükle (B) ← proc t (B)		
k yığınına geç (B)		
tuzaktan dönüş (B'ye)**(return-from-trap into B)**		

reg'leri geri yükle (B) ← k yığını (B)

kullanıcı moduna geç

B'nin bilgisayarına atla

Süreç B

...

Şekil 6.3: Sınırlı Doğrudan Yürütme Protokolü (Zamanlayıcı Kesintisi)

(Limited Direct Execution Protocol (Timer Interrupt))

6.4 Eşzamanlılık Konusunda Endişeli misiniz?

Dikkatli ve düşünceli okuyucular olarak bazılarınız şimdi düşünüyor olabilir: "Hmm... bir sistem çağrısı sırasında bir zamanlayıcı kesintisi meydana geldiğinde ne olur?" veya "Bir kesmeyi ele alırken diğeri olduğunda ne olur? Çekirdekte bunu halletmek zor olmuyor mu?"İyi sorular - henüz sizin için gerçekten umudumuz var!

Cevap evet, işletim sisteminin, kesme veya tuzak işleme sırasında başka bir kesme meydana gelirse ne olacağı konusunda gerçekten endişelenmesi gerekiyor. Bu, aslında, bu kitabın ikinci parçasının tamamının eşzamanlılık**(concurrency)** konusundaki tam konusudur; O zamana kadar ayrıntılı bir tartışmayı erteleyeceğiz.

İştahınızı kabartmak için, işletim sisteminin bu zor durumlarla nasıl başa çıktığına dair bazı temel bilgileri çizeceğiz. Bir işletim sisteminin yapabileceği basit bir şey, kesme işlemi sırasında kesintileri devre dışı bırakmaktır**(disable interrupts)**; Bunu yapmak, ne zaman

```
1 # void swtch (yapı bağlamı ** eski, yapı bağlamı * yeni);
2 #
3 # Mevcut kayıt bağlamını eskiye kaydet
4 # ve ardından kayıt bağlamını yeni'den yükleyin.
5 .küresel anahtar
6 anahtarı:
7 # Eski kayıtları kaydet
8 movl 4 (% esp),% eax # eski ptr'yi eax'a yerleştirin
9 popl 0 (%eax) # eski IP'yi kaydet
10 movl% esp, 4 (% eax) # ve yığın
11 movl% ebx, 8 (% eax) # ve diğer kayıtlar
12 movl %eax, 12 (%eax)
13 movl %edx, 16 (%eax)
14 movl %esı, 20 (%eax)
15 movl %edı, 24 (%eax)
16 movl %ebp, 28 (%eax)
17
```

```
18 # Yeni kayıtlar yükle
19 movl 4 (% esp), % eax # eax'a yeni ptr koy
20 movl 28 (% eax), %ebp # diğer kayıtları geri yükle
21 movl 24(%eax), %edı
22 movl 20(%eax), %esı
23 movl 16(%eax), %edx
24 movl 12(%eax), %ecx
25 movl 8(%eax), %ebx
26 movl 4 (% eax), %esp # yığını burada değiştirildi
27 pushl 0 (%eax) # iade adresi yerine kondu
28 ret # sonunda yeni ctxt'ye geri dönün
```

Şekil 6.4: xv6 Bağlam Değiştirme Kodu

(The xv6 Context Switch Code)

bir kesme işleniyor, başka hiç kimse CPU'ya teslim edilmeyecek. Tabii ki, işletim sisteminin bunu yaparken dikkatli olması gerekir; kesintileri çok uzun süre devre dışı bırakmak, (teknik açıdan) kötü olan kesintilerin kaybolmasına neden olabilir.

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

6.5 Özet

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

BİR KENARA: BAĞLAM ANAHTARLARI NE KADAR SÜRER

Sahip olabileceğiniz doğal bir soru şudur: bağlam değiştirme gibi bir şey ne kadar sürer? Hatta bir sistem çağrısı mı? Değerli olanlarınız için, tam olarak bunları ölçen **lmbench** [MS96] adlı bir araç ve ilgili olabilecek diğer birkaç performans önlemi vardır.

Sonuçlar zaman içinde oldukça gelişti ve kabaca işlemci performansını izledi. Örneğin, 1996'da Linux 1.3.37'yi 200 mhz'lik bir P6 cpu'da çalıştıran sistem çağrıları kabaca 4 mikrosaniye ve bir bağlam anahtarı kabaca 6 mikrosaniye aldı [MS96]. Modern sistemler, 2 veya 3 GHz işlemcili sistemlerde mikrosaniyenin

altında sonuçlarla neredeyse bir or- der büyüklüğünde daha iyi performans gösterir.

Tüm işletim sistemi eylemlerinin İŞLEMCİ başına performansı izlemediğine dikkat edilmelidir. Ousterhout'un gözlemlediği gibi, birçok işletim sistemi işlemi bellek yoğundur ve bellek bant genişliği zaman içinde işlemci hızı kadar önemli ölçüde iyileşmemiştir [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 uygulanmaktadır. Örneğin, çocuğu olan veya en azından çocukları duymuş olanlarınız, bebek odası sağlama**(baby proofing)** kavramına aşina olabilir: dolapları kilitlemek, tehlikeli maddeleri kirletmek ve elektrik prizlerini kapatmak. Oda bu şekilde hazırlandığında, odanın en tehlikeli yönlerinin kısıtlandığını bilerek bebeğinizin özgürce dolaşmasına izin verebilirsiniz.

Benzer şekilde, işletim sistemi CPU'yu önce (önyükleme süresini kısaltarak) tuzak işleyicilerini kurarak ve bir kesme zamanlayıcısını başlatarak ve ardından yalnızca işlemleri sınırlı bir modda çalıştırarak ”bebek kanıtlar". Bunu yaparak işletim sistemi, işlemlerin verimli çalışabileceğinden, yalnızca ayrıcalıklı işlemleri gerçekleştirmek için işletim sistemi müdahalesi gerektirdiğinden veya CPU'yu çok uzun süre tekelleştirdiklerinde ve bu nedenle devre dışı bırakılmaları gerektiğinde oldukça emin olabilir.

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

BİR KENARA: TEMEL CPU SANALLAŞTIRMA TERİMLERİ (MEKANİZMALAR)

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

* Tipik kullanıcı uygulamaları kullanıcı modunda çalışır ve bir sistem çağrısı**(system call)** kullanır işletim sistemi hizmetleri istemek için çekirdeğe tuzak**(trap)** kurmak.

* Trap talimatı kayıt durumunu dikkatli bir şekilde kaydeder, donanım durumunu çekirdek moduna değiştirir ve işletim sistemine önceden belirlenmiş bir hedefe atlar: trap tablosu**(trap table)**.

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

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

• Bir program çalıştırıldıktan sonra, işletim sistemi, kullanıcı programının sonsuza kadar çalışmamasını sağlamak için donanım mekanizmalarını kullanmalıdır, yani zamanlayıcı kesintisi**(timer interrupt)**. Bu yaklaşım, CPU zamanlamasına işbirliğine dayalı olmayan**(non-cooperative)** bir yaklaşımdır.

* 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.

Referanslar**(References)**

[A79] Xerox'un “Alto Kullanıcı El Kitabı". Xerox Palo Alto Araştırma Merkezi, Eylül 1979. Mevcut: http://history-computer.com/Library/AltoUsersHandbook.pdf . Zamanının çok ilerisinde inanılmaz bir sistem. Ünlü oldu çünkü Steve Jobs ziyaret etti, not aldı ve Lisa'yı ve sonunda Mac'i kurdu.

[C +04] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, A. Fox'un “Microreboot — Ucuz Kurtarma Tekniği”. OSDİ '04, San Francisco, CA, Aralık 2004. Daha sağlam sistemler oluştururken yeniden başlatma ile ne kadar ileri gidilebileceğine işaret eden mükemmel bir makale.

[İ11] Cilt 3A ve 3b'den “Intel 64 ve IA-32 Mimarileri Yazılım Geliştirici Kılavuzu”: Sistem Programlama Kılavuzu. Intel Corporation, Ocak 2011. Bu sadece sıkıcı bir kılavuzdur, ancak bazen bunlar yararlıdır.

[K + 61] T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner tarafından ”Tek Seviyeli Depolama Sistemi". Elektronik Bilgisayarlarda IRE İşlemleri, Nisan 1962. Atlas, modern sistemlerde gördüğünüzün çoğuna öncülük etti. Ancak, bu makale okumak için en iyisi değil. Sadece bir tanesini okuyacak olsaydınız, aşağıdaki tarihsel perspektifi deneyebilirsiniz [L78].

[L78] “Manchester Mark I ve Atlas: Tarihsel Bir Bakış Açısı”, S. H. Lavington. Acm'nin birleşmeleri, 21:1, Ocak 1978. Bilgisayarların erken gelişiminin tarihi ve Atlas'ın öncü çabaları.

[M + 63] J. McCarthy, S. Boilen tarafından “Küçük Bir Bilgisayar için Zaman Paylaşımlı Hata Ayıklama Sistemi”,

E. Fredkin, J. C. R. Yalayıcı. AFIPS '63 (İlkbahar), Mayıs 1963, New York, ABD. Bir zamanlayıcı kesintisi kullanmayı ifade eden zaman paylaşımı hakkında erken bir makale; Bunu tartışan alıntı: "Kanal 17 saat rutininin temel görevi, geçerli kullanıcının çekirdekten kaldırılıp kaldırılmayacağına karar vermek ve eğer öyleyse, dışarı çıkarken hangi kullanıcı programını değiştireceğine karar vermektir.”

[MS96] Larry McVoy ve Carl Staelin'in “lmbench: Performans analizi için taşınabilir araçlar”. USENIX Yıllık Teknik Konferansı, Ocak 1996. İşletim sisteminiz ve performansı hakkında bir dizi farklı şeyin nasıl ölçüleceği hakkında eğlenceli bir makale. Lmbench'i indirin ve bir deneyin.

[M11] Apple Computer, Inc. tarafından “Mac OS 9”.. Ocak 2011. http://en.wikipedia.org/wiki / Mac OS 9 . İsterseniz muhtemelen bir OS 9 öykünücüsü bile bulabilirsiniz; kontrol et, bu bir

eğlenceli küçük Mac!

[O90] "İşletim Sistemleri Neden Donanım Kadar Hızlı Hızlanmıyor?" J. Ouster- hout tarafından. USENIX Yaz Konferansı, Haziran 1990. İşletim sistemi performansının doğası üzerine klasik bir makale.

[S10] Open Group'un “Tek UNİX Spesifikasyonu, Sürüm 3", Mayıs 2010. Mevcut: http://www.unix.org/version3 /. Bunu okumak zor ve acı vericidir, bu yüzden mümkünse muhtemelen bundan kaçının. Mesela, biri okuman için sana para ödemiyorsa. Ya da o kadar meraklısın ki yardım edemezsin!

[S07] Hovav Shacham'ın “Kemikteki Masum Etin Geometrisi: İşlev Çağrıları olmadan libc'ye Dönüş (x86'da)”. CCS '07, Ekim 2007. Zaman zaman araştırmalarda göreceğiniz harika, akıllara durgunluk

veren fikirlerden biri. Yazar, keyfi olarak koda atlayabilirseniz, istediğiniz herhangi bir kod dizisini (büyük bir kod tabanı göz önüne alındığında) bir araya getirebileceğinizi gösterir; ayrıntılar için makaleyi okuyun. Teknik, ne yazık ki kötü niyetli saldırılara karşı savunmayı daha da zorlaştırıyor.

Ödev (Ölçme) **(Homework(Measurement))**

BİR KENARA: ÖLÇÜM ÖDEVLERİ

Ölçüm ödevleri, işletim sistemi veya donanım performansının bir yönünü ölçmek için gerçek bir makinede çalışacak kod yazdığınız küçük alıştırmalardır. Bu tür ödevlerin arkasındaki fikir, size gerçek bir işletim sistemi ile biraz uygulamalı deneyim sunmaktır.

Bu ödevde, bir sistem çağrısı ve bağlam anahtarının maliyetlerini ölçeceksiniz. Bir sistem çağrısının maliyetini ölçmek nispeten kolaydır. Örneğin, basit bir sistem çağrısını (örneğin, 0 baytlık bir okuma gerçekleştirerek) ve ne kadar sürdüğünü art arda arayabilirsiniz; Zamanı yineleme sayısına bölmek size bir sistem çağrısının maliyetinin bir tahminini verir.

Dikkate almanız gereken bir şey, zamanlayıcınızın hassasiyeti ve açık saçıklığıdır. Kullanabileceğiniz tipik bir zamanlayıcı gettimeofday(); ayrıntılar için kılavuz sayfasını okuyun. Orada göreceğiniz şey, gettimeofday() öğesinin 1970'ten bu yana mikrosaniye cinsinden zamanı döndürmesidir; Ancak bu, zamanlayıcının mikrosaniyeye kadar kesin olduğu anlamına gelmez. Arka arkaya aramaları ölçün

zamanlayıcının yeniden müttefikinin ne kadar hassas olduğu hakkında bir şeyler öğrenmek için gettimeofday() ; 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 UNİX kanalı kurarak yapar; Bir kanal, bir UNİX sistemindeki işlemlerin birbirleriyle iletişim kurabilmesinin birçok yolundan sadece biridir. İlk işlem daha sonra ilk boruya bir yazma verir ve ikincisinde bir okuma bekler; ikinci borudan bir şeyin okunmasını bekleyen ilk işlemi gördükten sonra, işletim sistemi ilk işlemi engellenmiş duruma getirir ve ilk borudan okuyan diğer işleme geçer ve sonra ikincisine yazar. İkinci işlem ilk borudan tekrar okumaya çalıştığında bloke olur ve böylece ileri geri iletişim döngüsü devam eder. Bu şekilde iletişim kurmanın maliyetini tekrar tekrar ölçerek, lmbench bir bağlam anahtarının maliyetini iyi bir şekilde tahmin edebilir. Boruları veya belki de UNİX soketleri gibi başka bir iletişim mekanizmasını kullanarak burada benzer bir şeyi yeniden oluşturmayı deneyebilirsiniz.

Bağlam değiştirme maliyetini ölçmede bir zorluk, birden fazla cpu'lu sistemlerde ortaya çıkar; Böyle bir sistemde yapmanız gereken, bağlam değiştirme işlemlerinizin aynı işlemcide bulunduğundan emin olmaktır. Genellikle, çoğu işletim sisteminin bir işlemi kısmi bir işlemciye bağlamak için çağrıları vardır; Örneğin Linux'ta sched setaffinity () çağrısı aradığınız şeydir. Her iki işlemin de aynı işlemcide

olduğundan emin olarak, işletim sisteminin bir işlemi durdurup diğerini aynı cpu'da geri yükleme maliyetini ölçtüğünüzden emin olursunuz.