6

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 böyle devam eder. CPU'yu bu şekilde paylaştırarak 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 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 Sistemi	Program
entry for process list Allocate memory for programLoad program into memory Set up stack with argc/argv Clear registers	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

Figür 6.1: Doğrudan Yürütme Protokolü (Sınırsız)

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 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!

6.1 Sorun 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 G/Ç 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 kontrol eden bir dosya sistemi oluşturmak istiyorsak, herhangi bir kullanıcının diske G/Ç'ler vermesine izin veremeyiz; bunu yaparsak, bir işlem tüm diski okuyabilir veya yazabilir ve bu nedenle tüm korumalar kaybolur

Bu nedenle, benimsediğimiz yaklaşım, kullanıcı modu olarak bilinen yeni bir işlemci modunu tanıtmaktır; kullanıcı modunda çalışan kodun yapabilecekleri sınırlıdır. Örneğin, kullanıcı modunda çalışırken, bir işlem G/Ç 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 vardır. Bu modda çalışan kod, G/Ç 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, 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 tahsis etmek.

İPUCU: KORUMALI KONTROL AKTARIMI KULLANIN
Donanım, farklı yürütme modları sağlayarak işletim sistemine yardımcı olur. Kullanıcı modunda, uygulamaların donanım kaynaklarına tam erişimi yoktur. Çekirdek modunda, işletim sisteminin makinenin tüm kaynaklarına erişimi vardır. Çekirdeğe tuzak kurmak ve tuzaktan kullanıcı modu programlarına geri dönmek için özel talimatlar ve işletim sisteminin donanıma tuzak tablosunun bellekte nerede olduğunu söylemesine izin veren talimatlar da sağlanır.

hafıza. 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 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 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 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 Fikir1'dir. Bu nedenle çekirdek, bir tuzakta hangi kodun yürütüleceğini dikkatlice kontrol etmelidir. Çekirdek bunu önyükleme sırasında bir tuzak tablosu 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.

OS @ boot (kernel mode)	Hardware	
tuzak tablosunu başlat		
	adresini hatırla... sistem çağrısı işleyicisi	

OS @ run (kernel mode)	Hardware	Program (user mode)
İşlem listesi için giriş oluştur Program için bellek tahsis et Programı belleğe yükle argv ile kullanıcı yığınını ayarla Çekirdek yığınını reg/PC ile doldur **return-from-trap**		
	kayıtları geri yükle (çekirdek yığınından) kullanıcı moduna geç ana ekrana geç	
		Run main() ...
	kayıtları kaydet (çekirdek yığınına) çekirdek moduna geç tuzak işleyicisine atla	Çağrı sistemi çağrısı işletim sistemine tuzak
Tutma tuzağı Sistem çağrısı işini yap **return-from-trap**		
	kayıtları geri yükle (çekirdek yığınından) kullanıcı moduna geç tuzaktan sonra PC'ye atla	
		... return from main **trap** (via `exit()`)
İşlemin boş hafızası İşlem listesinden kaldır		

Şekil 6.2: **Sınırlı Doğrudan Yürütme Protokolü**

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

İPUCU: GÜVENLİ SİSTEMLERDE KULLANICI GİRİŞLERİNE 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 bir işletim sistemi uygulamanın hala birçok yönü vardır. düşünmeliyiz. Bunlardan biri, sistem çağrısı sınırında bağımsız değişkenlerin işlenmesidir; işletim sistemi, kullanıcının ne ilettiğini kontrol etmeli ve bağımsız değişkenlerin uygun şekilde belirtildiğinden emin olmalı veya başka bir şekilde aramayı reddetmelidir.

Örneğin, bir write() sistem çağrısıyla, kullanıcı yazma çağrısının kaynağı olarak bir arabelleğin adresini belirtir. Kullanıcı (yanlışlıkla veya kötü niyetle) "kötü" bir adres girerse (örneğin, adres alanının çekirdeğin bölümündeki bir adres), 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 herhangi bir işlemin belleğini okumasını sağlar.

Kesin sistem çağrısını belirtmek için, genellikle her sistem çağrısına bir sistem çağrı numarası atanır. Dolayısıyla kullanıcı kodu, istenen sistem çağrı numarasını bir kayda veya yığın üzerinde belirli bir yere yerleştirmekten sorumludur; İşletim sistemi tuzak işleyicisi içinde sistem çağrısını işlerken bu numarayı inceler, geçerli olduğundan emin olur ve geçerliyse ilgili kodu yürütür. Bu düzeydeki dolaylılık, bir koruma biçimi olarak hizmet eder; kullanıcı kodu, atlamak için tam bir adres belirtemez, bunun yerine numara aracılığıyla belirli bir hizmet talep etmelidir.

Son olarak bir kenara: donanıma tuzak tablolarının nerede olduğunu söyleyen talimatı uygulayabilmek çok güçlü bir yetenektir. Dolayısıyla tahmin edebileceğiniz gibi ayrıcalıklı bir işlemdir. Bu talimatı kullanıcı modunda çalıştırmaya çalışırsanız, donanım size izin vermez ve muhtemelen ne olacağını tahmin edebilirsiniz (ipucu: adios, rahatsız edici program). Düşünmek için gelin: Kendi tuzak tablonuzu kurabilseydiniz, bir sisteme ne gibi korkunç şeyler yapabilirdiniz? Makineyi devralabilir misin?

Zaman çizelgesi (Şekil 6.2'de aşağı doğru artan süre ile) protokolü özetlemektedir. Her işlemin, çekirdeğe girip çıkarken kayıtların (genel amaçlı kayıtlar ve program sayacı dahil) kaydedildiği ve çekirdeğe (donanım tarafından) geri yüklendiği bir çekirdek yığını 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 yönerge aracılığıyla yapar (tüm ayrıcalıklı yönergelerkoyurenklevurgulanmıştır).

6.1 İkinci aşamada (bir işlemi çalıştırırken), işlemin yürütülmesini başlatmak için bir tuzaktan dönüş yönergesini kullanmadan önce çekirdek birkaç şeyi ayarlar (örneğin, işlem listesinde bir düğüm tahsis etmek, bellek ayırmak); bu, CPU'yu kullanıcı moduna geçirir ve işlemi çalıştırmaya başlar. Süreç bir sistem çağrısı yapmak istediğinde, onu işleyen işletim sistemine geri döner ve bir kez daha tuzaktan dönüş yoluyla sürece kontrolü geri verir. İşlem daha sonra işini tamamlar ve main();'den döner. bu genellikle programdan düzgün bir şekilde çıkacak olan bir saplama koduna geri döner (örneğin, işletim sistemine tuzak kuran exit() sistem çağrısını çağırarak). Bu noktada, işletim sistemi temizlenir ve işimiz biter.

6.2 Problem #2: Süreçler Arasında Geçiş

Doğrudan yürütme ile ilgili bir sonraki sorun, süreçler arasında geçiş yapmaktır. İşlemler arasında geçiş yapmak basit olmalı, değil mi? İşletim sistemi yalnızca bir işlemi durdurmaya ve başka bir işlemi başlatmaya karar vermelidir. Problem ne? Ama aslında biraz aldatıcıdır: özellikle, CPU üzerinde bir işlem çalışıyorsa, bu tanım gereği işletim sisteminin çalışmadığı anlamına gelir. İşletim sistemi çalışmıyorsa, herhangi bir şeyi nasıl yapabilir? (ipucu: yapamaz) Bu neredeyse felsefi görünse de, gerçek bir sorundur: CPU üzerinde çalışmıyorsa işletim sisteminin herhangi bir işlem yapması açıkça mümkün değildir. Böylece sorunun can alıcı noktasına geliyoruz.

EN ÖNEMLİ NOKTA: CPU KONTROLÜ NASIL YENİDEN ELDE EDİLİR?
İşletim sistemi, işlemler arasında geçiş yapabilmek için CPU'nun kontrolünü nasıl geri alabilir?

İşbirlikçi Bir Yaklaşım: Sistem Çağrılarını Bekleyin

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

Dolayısıyla, bu ütopik dünyada dostça bir süreç CPU'dan nasıl vazgeçer diye sorabilirsiniz. Görünüşe göre çoğu işlem, örneğin bir dosyayı açıp ardından okumak veya başka bir makineye mesaj göndermek veya yeni bir işlem oluşturmak için sistem çağrıları yaparak CPU'nun kontrolünü işletim sistemine oldukça sık aktarıyor. . Bunun gibi sistemler genellikle, diğer işlemleri çalıştırabilmesi için kontrolü işletim sistemine aktarmak dışında hiçbir şey yapmayan açık bir verim sistem çağrısını içerir.

Uygulamalar ayrıca yasa dışı bir şey yaptıklarında kontrolü işletim sistemine aktarırlar.

Bu protokol sırasında meydana gelen iki tür kayıt kaydetme/geri yükleme olduğunu unutmayın. İlki, zamanlayıcı kesintisinin meydana geldiği zamandır; bu durumda, çalışan işlemin kullanıcı kayıtları, o işlemin çekirdek yığınını kullanan donanım tarafından dolaylı olarak kaydedilir. İkincisi, işletim sisteminin A'dan B'ye geçmeye karar verdiği zamandır; bu durumda, çekirdek kayıtları yazılım (yani işletim sistemi) tarafından açıkça kaydedilir, ancak bu sefer sürecin süreç yapısındaki belleğe kaydedilir. İkinci eylem, sistemi A'dan çekirdeğe yeni sıkışmış gibi çalışmaktan B'den çekirdeğe yeni sıkışmış gibi çalıştırır.

İşbirlikçi Olmayan Bir Yaklaşım: İşletim Sistemi Kontrolü Ele Geçiriyor

Donanımdan bazı ek yardımlar olmadan, bir süreç sistem çağrıları (veya hatalar) yapmayı reddettiğinde ve böylece kontrolü işletim sistemine geri verdiğinde, işletim sisteminin pek bir şey yapamayacağı ortaya çıktı. Aslında, işbirlikçi yaklaşımda, bir süreç sonsuz bir döngüde sıkışıp kaldığında tek başvurunuz, bilgisayar sistemlerindeki tüm sorunlara asırlık çözüme başvurmak: makineyi yeniden başlatmak. Böylece, yine CPU'nun kontrolünü ele geçirmeye yönelik genel arayışımızın bir alt problemine ulaşıyoruz.

EN ÖNEMLİ NOKTA: İŞBİRLİĞİ OLMADAN KONTROL NASIL ELDE EDİLİR?

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

Cevabın basit olduğu ortaya çıktı ve yıllar önce bilgisayar sistemleri kuran Birkaç kişi tarafından keşfedildi:bir zamanlayıcı kesintisi [M+63]. Bir Zamanlayıcı cihazı her,milisaniyede bir kesinti oluşturacak şekilde programlanabilir; kesme yükseltildiğinde, o anda çalışmakta olan işlem durdurulur ve işletim sisteminde önceden yapılandırılmış bir kesme işleyicisi çalışır. Bu noktada, işletim sistemi CPU'nun kontrolünü yeniden ele geçirdi ve böyleyece canının istediğini yapabilir;mevcut işlemi durdurun ve farklı bir işlem başlatın.

> Daha önce sistem çağrılarında tartıştığımız gibi, işletim sistemi zamanlayıcı kesintisi meydana geldiğinde hangi kodun çalıştırılacağını donanıma bildirmelidir; bu nedenle, önyükleme sırasında işletim sistemi tam olarak bunu yapar. İkinci olarak, önyükleme sırasında da işletim sisteminin zamanlayıcıyı başlatması gerekir ki bu elbette ayrıcalıklı bir ayardır.

İPUCU: UYGULAMANIN YANLIŞ DAVRANIŞLARIYLA BAŞA ÇIKMA
İşletim sistemleri genellikle tasarım (kötü niyetlilik) veya kaza (hata) yoluyla yapmamaları gereken bir şeyi yapmaya çalışan yanlış davranan süreçlerle uğraşmak zorundadır. Modern sistemlerde, İşletim Sisteminin bu tür suiistimali halletmeye çalıştığı yol, basitçe suçluyu sonlandırmaktır. Bir vuruş ve sen dışarıdasın! Belki acımasız olabilir, ancak belleğe yasa dışı olarak erişmeye çalıştığınızda veya yasa dışı bir talimat yürüttüğünüzde işletim sistemi başka ne yapmalıdır?

operasyon. Zamanlayıcı başladıktan sonra, işletim sistemi, kontrolün sonunda kendisine iade edileceği konusunda kendini güvende hissedebilir ve böylece işletim sistemi, kullanıcı programlarını çalıştırmakta serbesttir. Zamanlayıcı da kapatılabilir (aynı zamanda ayrıcalıklı bir işlem), eşzamanlılığı daha ayrıntılı olarak anladığımızda daha sonra tartışacağımız bir şey.Bir kesinti meydana geldiğinde, özellikle kesinti meydana geldiğinde çalışmakta olan programın durumunu, sonraki bir tuzaktan dönüş komutunun çalışan programı devam ettirebilmesi için yeterince kaydetme konusunda donanımın bazı sorumlulukları olduğunu unutmayın. doğru şekilde. Bu eylemler dizisi, çekirdeğe açık bir sistem çağrısı tuzağı sırasında donanımın davranışına oldukça benzerdir; çeşitli kayıtlar böylece kaydedilir (örneğin, bir çekirdek yığınına) ve böylece tuzaktan dönüş talimatı tarafından kolayca geri yüklenir .

Bağlamı Kaydetme ve Geri Yükleme

Artık işletim sistemi, ister bir sistem çağrısı yoluyla iş birliği içinde isterse bir zamanlayıcı kesintisi yoluyla daha güçlü bir şekilde kontrolü yeniden ele aldığına göre, bir karar verilmelidir: o anda çalışan işlemi çalıştırmaya devam etmek veya farklı bir işleme geçmek. Bu karar, işletim sisteminin zamanlayıcı olarak bilinen bir bölümü tarafından verilir; sonraki birkaç bölümde çizelgeleme politikalarını ayrıntılı olarak tartışacağız.

Değiştirme kararı verilirse, işletim sistemi daha sonra bağlam anahtarı olarak adlandırdığımız düşük seviyeli bir kod parçasını yürütür. Bir bağlam anahtarı kavramsal olarak basittir: İşletim sisteminin tek yapması gereken, o anda yürütülen işlem için birkaç kayıt değeri kaydetmek (örneğin, çekirdek yığınına) ve yakında yürütülecek olan işlem için birkaçını geri yüklemektir (çekirdek yığını). Bunu yaparak, işletim sistemi, tuzaktan dönüş talimatı nihayet yürütüldüğünde, çalışmakta olan işleme geri dönmek yerine, sistemin başka bir işlemi yürütmeye devam etmesini sağlar.

Halihazırda çalışan işlemin içeriğini kaydetmek için, işletim sistemi, genel amaçlı kayıtları, PC'yi ve o anda çalışan işlemin çekirdek yığını işaretçisini kaydetmek için bazı düşük seviyeli derleme kodlarını yürütecek ve ardından söz konusu işlemi geri yükleyecektir. kaydeder, PC ve yakında yürütülecek olan işlem için çekirdek yığınına geçin. Yığınları değiştirerek çekirdek, bir işlem (kesintiye uğrayan) bağlamında anahtar kodu çağrısını girer ve bir başkası (yakında yürütülecek olan) bağlamında geri döner. İşletim sistemi en sonunda bir tuzaktan dönüş talimatını yürüttüğünde,

İPUCU: KONTROLÜ YENİDEN ELDE ETMEK İÇİN ZAMANLAYICI KESMEYİ KULLANIN

Bir zamanlayıcı kesintisinin eklenmesi, işlemler işbirlikçi olmayan bir şekilde hareket etse bile işletim sistemine bir CPU üzerinde yeniden çalışma yeteneği verir. Bu nedenle, bu donanım özelliği, işletim sisteminin makinenin kontrolünü sürdürmesine yardımcı olmak için gereklidir.

İPUCU: YENİDEN BAŞLATMA YARARLIDIR

Daha önce, işbirlikçi önleme altında sonsuz döngülere (ve benzer davranışlara) yönelik tek çözümün makineyi yeniden başlatmak olduğunu belirtmiştik. Bu hack'le dalga geçseniz de, araştırmacılar, yeniden başlatmanın (veya genel olarak, bir yazılım parçası üzerinden başlamanın) sağlam sistemler oluşturmak için son derece yararlı bir araç olabileceğini göstermiştir [C+04].

Özellikle yeniden başlatma, yazılımı bilinen ve muhtemelen daha test edilmiş bir duruma geri taşıdığı için yararlıdır. Yeniden başlatmalar ayrıca başka türlü idare edilmesi zor olabilecek eski veya sızan kaynakları (örn. bellek) geri alır. Son olarak, yeniden başlatmaların otomatikleştirilmesi kolaydır. Tüm bu nedenlerden dolayı, büyük ölçekli küme İnternet hizmetlerinde, sistem yönetimi yazılımının makine setlerini sıfırlamak ve böylece yukarıda listelenen avantajları elde etmek için periyodik olarak yeniden başlatması alışılmadık bir durum değildir.

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

yakında yürütülecek olan süreç, şu anda çalışan süreç haline gelir. Ve böylece bağlam anahtarı tamamlandı.

Tüm sürecin bir zaman çizelgesi Şekil 6.3'te gösterilmektedir. Bu örnekte, İşlem A çalışıyor ve ardından zamanlayıcı kesmesi tarafından kesintiye uğratılıyor. Donanım, kayıtlarını (çekirdek yığınına) kaydeder ve çekirdeğe girer (çekirdek moduna geçer). Zamanlayıcı kesme işleyicisinde, işletim sistemi, İşlem A'yı çalıştırmaktan İşlem B'ye geçmeye karar verir. Bu noktada, mevcut kayıt değerlerini (A'nın işlem yapısına) dikkatli bir şekilde kaydeden ve kayıtlarını geri yükleyen switch() yordamını çağırır. İşlem B'yi (işlem yapısı girişinden) ve ardından, özellikle B'nin çekirdek yığınını (A'nın değil) kullanmak için yığın işaretçisini değiştirerek bağlamları değiştirir. Son olarak, OS, B'nin kayıtlarını geri yükleyen ve onu çalıştırmaya başlayan tuzaktan geri döner.

Bu protokol sırasında meydana gelen iki tür kayıt kaydetme/geri yükleme olduğunu unutmayın. İlki, zamanlayıcı kesintisinin meydana geldiği zamandır; bu durumda, çalışan işlemin kullanıcı kayıtları, o işlemin çekirdek yığınını kullanan donanım tarafından dolaylı olarak kaydedilir. İkincisi, işletim sisteminin A'dan B'ye geçmeye karar verdiği zamandır; bu durumda, çekirdek kayıtları yazılım (yani işletim sistemi) tarafından açıkça kaydedilir, ancak bu sefer sürecin süreç yapısındaki belleğe kaydedilir. İkinci eylem, sistemi A'dan çekirdeğe yeni sıkışmış gibi çalışmaktan B'den çekirdeğe yeni sıkışmış gibi çalıştırır.

yakında yürütülecek olan süreç, şu anda çalışan süreç haline gelir. Ve böylece bağlam anahtarı tamamlandı.

Tüm sürecin bir zaman çizelgesi Şekil 6.3'te gösterilmektedir. Bu örnekte, İşlem A çalışıyor ve ardından zamanlayıcı kesmesi tarafından kesintiye uğratılıyor. Donanım, kayıtlarını (çekirdek yığınına) kaydeder ve çekirdeğe girer (çekirdek moduna geçer). Zamanlayıcı kesme işleyicisinde, işletim sistemi, İşlem A'yı çalıştırmaktan İşlem B'ye geçmeye karar verir. Bu noktada, mevcut kayıt değerlerini (A'nın işlem yapısına) dikkatli bir şekilde kaydeden ve kayıtlarını geri yükleyen switch() yordamını çağırır. İşlem B'yi (işlem yapısı girişinden) ve ardından, özellikle B'nin çekirdek yığınını (A'nın değil) kullanmak için yığın işaretçisini değiştirerek bağlamları değiştirir. Son olarak, OS, B'nin kayıtlarını geri yükleyen ve onu çalıştırmaya başlayan tuzaktan geri döner.

Bu protokol sırasında meydana gelen iki tür kayıt kaydetme/geri yükleme olduğunu unutmayın. İlki, zamanlayıcı kesintisinin meydana geldiği zamandır; bu durumda, çalışan işlemin kullanıcı kayıtları, o işlemin çekirdek yığınını kullanan donanım tarafından dolaylı olarak kaydedilir. İkincisi, işletim sisteminin A'dan B'ye geçmeye karar verdiği zamandır; bu durumda, çekirdek kayıtları yazılım (yani işletim sistemi) tarafından açıkça kaydedilir, ancak bu sefer sürecin süreç yapısındaki belleğe kaydedilir. İkinci eylem, sistemi A'dan çekirdeğe yeni sıkışmış gibi çalışmaktan B'den çekirdeğe yeni sıkışmış gibi çalıştırır.

Böyle bir anahtarın nasıl devreye girdiğini daha iyi anlamanız için, Şekil 6.4'te xv6 için bağlam anahtarı kodu gösterilmektedir. Bir anlam ifade edip edemeyeceğinize bakın (bunu yapmak için biraz x86 ve ayrıca biraz xv6 bilmeniz gerekecek). Eski ve yeni bağlam yapıları sırasıyla eski ve yeni sürecin süreç yapılarında bulunur.

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember addresses of...
	syscall handler
	timer handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) → k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) → proc-t(A)		
restore regs(B) ← proc t(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) ← k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

Figure 6.3: **Limited Direct Execution Protocol (Timer Interrupt)**

6.3 Eşzamanlılık Konusunda Endişeli misiniz?

Dikkatli ve düşünceli okuyucular olarak bazılarınız şimdi şöyle düşünüyor olabilir: "Hmm... bir sistem çağrısı sırasında bir zamanlayıcı kesintisi meydana geldiğinde ne olur?" ya da "Bir kesintiyle uğraşırken diğeri olduğunda ne olur? Bunu çekirdekte halletmek zor olmuyor mu?" İyi sorular - sizin için gerçekten biraz umudumuz var!

Yanıt evettir, kesinti veya tuzak yönetimi sırasında başka bir kesinti meydana gelirse işletim sisteminin gerçekten de ne olacağı konusunda endişelenmesi gerekir. Bu, aslında, bu kitabın eşzamanlılık üzerine olan ikinci bölümünün tam konusu; ayrıntılı bir tartışmayı o zamana kadar erteleyeceğiz.

İştahınızı kabartmak için, işletim sisteminin bu zor durumlarla nasıl başa çıktığına dair bazı temel bilgileri ele alacağız. Bir işletim sisteminin yapabileceği basit bir şey, kesinti işleme sırasında kesintileri devre dışı bırakmaktır; bunu yapmak, ne zaman

```
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
```

Figure 6.4: **The xv6 Context Switch Code**

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

6.4 İşletim sistemleri ayrıca dahili veri yapılarına eşzamanlı erişimi korumak için bir dizi karmaşık kilitleme şeması geliştirmiştir. Bu, çekirdek içinde aynı anda birden çok etkinliğin devam etmesini sağlar, özellikle çok işlemcililerde yararlıdır. Eşzamanlılıkla ilgili bu kitabın bir sonraki bölümünde göreceğimiz gibi, bu tür bir kilitleme karmaşık olabilir ve çeşitli ilginç ve bulunması zor hatalara yol açabilir

6.5 Özet

Toplu olarak sınırlı doğrudan yürütme olarak adlandırdığımız bir dizi teknik olan CPU sanallaştırmayı uygulamak için bazı temel düşük seviyeli mekanizmaları tanımladık. Temel fikir basittir: sadece CPU'da çalıştırmak istediğiniz programı çalıştırın, ancak önce donanımı kurduğunuzdan emin olun.

işlemin işletim sistemi yardımı olmadan yapabileceklerini sınırlamak için.

KENARA: BAĞLAM DEĞİŞİMLERİ NE KADAR SÜRER

Aklınıza gelebilecek doğal bir soru şudur: Bağlam değişikliği gibi bir şey ne kadar sürer? Ya da bir sistem çağrısı? Merak edenler için, tam olarak bu şeyleri ölçen lmbench [MS96] adlı bir araç ve ayrıca ilgili olabilecek birkaç başka performans ölçüsü var.

Sonuçlar, kabaca işlemci performansını izleyerek zaman içinde oldukça iyileşti. Örneğin, 1996'da 200 MHz'lik bir P6 CPU'da Linux 1.3.37 çalıştırırken, sistem çağrıları yaklaşık 4 mikrosaniye ve bir içerik değiştirme işlemi yaklaşık 6 mikrosaniye [MS96] sürmüştür. Modern sistemler, 2 veya 3 GHz işlemcili sistemlerde mikrosaniyenin altında sonuçlarla neredeyse kat kat daha iyi performans gösteriyor.

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 odayı bebek korumalı hale getirme kavramına aşina olabilir: tehlikeli maddeler içeren dolapları kilitleme ve elektrik prizlerini kapatma. Oda bu şekilde hazır olduğunda, bebeğinizin odanın en tehlikeli kısımlarının kısıtlandığını bilerek özgürce dolaşmasına izin verebilirsiniz.

Benzer bir şekilde, OS önce (önyükleme sırasında) tuzak işleyicileri ayarlayarak ve bir kesme zamanlayıcısı başlatarak ve ardından işlemleri yalnızca kısıtlı bir modda çalıştırarak CPU'yu "bebek korumalı" yapar. Bunu yaparak, işletim sistemi, ayrıcalıklı işlemleri gerçekleştirmek için yalnızca işletim sistemi müdahalesi gerektirdiğinde veya CPU'yu çok uzun süre tekelleştirdiklerinde ve bu nedenle devre dışı bırakılmaları gerektiğinde, işlemlerin verimli bir şekilde çalışabileceğinden oldukça emin olabilir.

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

KENARA: ANAHTAR CPU SANALLATMA ŞARTLARI (MEKANİZMALAR)

- CPU en az iki yürütme modunu desteklemelidir: kısıtlı kullanıcı modu ve ayrıcalıklı (kısıtlanmamış) çekirdek modu.
  Tipik kullanıcı uygulamaları, kullanıcı modunda çalışır ve bir sistem çağrısı kullanır işletim sistemi hizmetleri istemek için çekirdeğe tuzak kurmak.
- Tuzak talimatı, kayıt durumunu dikkatli bir şekilde kaydeder, donanım durumunu çekirdek moduna değiştirir ve işletim sisteminde önceden belirlenmiş bir hedefe atlar: tuzak tablosu.
- İşletim sistemi bir sistem çağrısına hizmet vermeyi bitirdiğinde, ayrıcalığı azaltan ve denetimi işletim sistemine atlayan tuzaktan sonra yönergeye geri döndüren başka bir özel tuzaktan dönüş talimatı aracılığıyla kullanıcı programına geri döner.
- Tuzak tabloları işletim sistemi tarafından önyükleme sırasında kurulmalı ve kullanıcı programları tarafından kolaylıkla değiştirilemediğinden emin olunmalıdır. Tüm bunlar, programları verimli bir şekilde ancak işletim sistemi kontrolünü kaybetmeden çalıştıran sınırlı doğrudan yürütme protokolünün bir parçasıdır.
- Bir program çalışırken, işletim sistemi, kullanıcı programının sonsuza kadar çalışmamasını sağlamak için donanım mekanizmalarını, yani zamanlayıcı kesintisini kullanmalıdır. Bu yaklaşım, CPU zamanlaması için işbirlikçi olmayan bir yaklaşımdır.
- Bazen işletim sistemi, bir zamanlayıcı kesintisi veya sistem çağrısı sırasında, mevcut işlemi çalıştırmaktan farklı bir işleme, bağlam anahtarı olarak bilinen düşük seviyeli bir tekniğe geçmek isteyebilir..

References

[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 ofthe 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. Ouster-hout. 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(ÖLÇME)

ÖLÇME ÖDEVLERİ

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

Bu ödevde, bir sistem çağrısının ve bağlam değişikliğinin maliyetlerini ölçeceksiniz. Bir sistem çağrısının maliyetini ölçmek nispeten kolaydır. Örneğin, basit bir sistem çağrısını (örneğin, 0 baytlık bir okuma gerçekleştirme) ve bunun ne kadar sürdüğünü tekrar tekrar çağırabilirsiniz; süreyi yineleme sayısına bölmek, size bir sistem çağrısının tahmini maliyetini verir.

Dikkate almanız gereken bir şey, zamanlayıcınızın kesinliği ve doğruluğudur. Kullanabileceğiniz tipik bir zamanlayıcı gettimeofday();'dir. ayrıntılar için man sayfasını okuyun. Orada göreceğiniz şey, gettimeofday()'in 1970'ten bu yana mikrosaniye cinsinden zamanı döndürmesidir; ancak bu, zamanlayıcının mikrosaniye hassasiyetinde olduğu anlamına gelmez. Arka arkaya aramaları ölçün

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

Bağlam anahtarının maliyetini ölçmek biraz daha zordur. Lbench kıyaslaması, bunu tek bir CPU üzerinde iki işlem çalıştırarak ve aralarında iki UNIX hattı kurarak yapar; boru, bir UNIX sistemindeki süreçlerin birbiriyle iletişim kurabileceği birçok yoldan yalnızca biridir. İlk işlem daha sonra birinci boruya bir yazma gönderir ve ikincide bir okuma bekler; ilk işlemin ikinci borudan bir şey okumak için beklediğini görünce, işletim sistemi ilk işlemi bloke durumuna alır ve ilk borudan okuyan ve ardından ikinciye yazan diğer işleme geçer. İkinci işlem birinci borudan tekrar okumaya çalıştığında bloke olur ve böylece iletişimin ileri geri döngüsü devam eder. Bu şekilde iletişim kurmanın maliyetini tekrar tekrar ölçerek, lmbench bir bağlam değişikliğinin maliyetine ilişkin iyi bir tahminde bulunabilir. Boruları veya UNIX yuvaları gibi başka bir iletişim mekanizmasını kullanarak burada benzer bir şeyi yeniden yaratmayı deneyebilirsiniz..

Bağlam değiştirme maliyetinin ölçülmesindeki bir zorluk, birden fazla CPU'ya sahip sistemlerde ortaya çıkar; böyle bir sistemde yapmanız gereken, bağlam değiştirme işlemlerinizin aynı işlemci üzerinde bulunmasını sağlamaktır. Şans eseri, çoğu işletim sisteminde bir işlemi belirli bir işlemciye bağlamak için çağrılar bulunur; örneğin, Linux'ta aradığınız şey sched setaffinity() çağrısıdır. Her iki işlemin de aynı işlemcide olmasını sağlayarak, işletim sisteminin bir işlemi durdurup diğerini aynı CPU'ya geri yüklemesinin maliyetini ölçtüğünüzden emin olursunuz.

ÖDEV ÇÖZÜMÜ

Context switch (içerik değiştirme) maliyet belirleme

Içerik değişikliğinin dolaylı maliyetini ölçmek zorlu bir sorundur.
sentetik bir iş yükü kullanarak bağlam değişikliğinin dolaylı maliyetinin deneysel olarak ölçülmesi. Spesifik olarak,
program veri boyutunun ve erişim adımının bağlam değiştirme maliyeti üzerindeki etkisi. Ayrıca işletim sisteminin potansiyel etkisini de gösteriyoruz ölçüm doğruluğu üzerinde arka plan kesme işleme. Bu etki, çok işlemcili bir işlemci kullanılarak hafifletilebilir.
Bir işlemcinin içerik değiştirme ölçümü için kullanıldığı, diğerinin ise işletim sistemi arka plan görevlerini çalıştırdığı sistem.

measureSwitch.c -- bir boru aracılığıyla iki işlem iletişimi
measureSingle.c -- iki işlem iletişimini simüle eden tek işlem
util.c -- bağlam değiştirme maliyeti ölçümü için yardımcı program
util.h -- yardımcı program başlık dosyası

Kullanım:

./measureSingle [-n ArraySize] [-s StrideSize]
./measureSwitch [-n ArraySize] [-s StrideSize]

Varsayılan olarak, ArraySize ve StrideSize'ın her ikisi de sıfırdır.
Hem ArraySize hem de StrideSize bayt cinsinden verilmelidir.

Çıktı:
MeasureSingle çıkış süresi1
time1 = Dizi boyunca geçiş yükü + boru yükü

MeasureSwitch çıkış süresi2
time2 = time1 + içerik değiştirme ek yükü

bağlam anahtarının toplam maliyeti = time2 - time1 (mikrosaniye)

bağlam değişikliğinin dolaylı maliyeti = toplam maliyet - doğrudan maliyet

Not:

Her iki parametre de sıfır olduğunda, sonuç (zaman2 - zaman1)
bağlam anahtarının doğrudan maliyeti.

MeasureSingle ve measureSwitch'i iki programa ayırıyoruz.
Birbirlerine neden olabilecek etkileşimi önlemek için.

measureSingle.c

```
/*
* AÇIKLAMA: İki işlem iletişimini simüle eden tek işlem programı
* ÇIKIŞ: time1 = Dizi boyunca geçiş yükü + boru yükü
* time2 almak için measureSwitch'i kullanılır
* içerik değiştirme maliyeti = time2 - time1
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <pthread.h>
#include <sys/time.h>
#include <ctype.h>
#include <math.h>
#include <string.h>
#include <sched.h>
#include <errno.h>
#include <linux/unistd.h>
#include "util.h"
```

mesasureSingle.c'nin devamı

```
void  showUsage ()
{
 fprintf ( stderr, " Kullanım: \n measureSingle <seçenekler> \n \
   -n <sayı> üzerinde çalışılacak dizinin boyutu (bayt olarak). varsayılan 0. \n \
   -s <sayı> erişim adım boyutu (bayt cinsinden). default 0 \n " );
}

void  measureSingle ( int dizi_boyutunu  kaydet, int adımını   kaydet ,
            int *p1 kaydı, char * msg kaydı, çift  * f kaydı ){
   int  i, j, m'yi kaydedin ;

   for ( i= 0 ; i<LOOP; i++) {
      for ( m= 0 ; m<adım; m++)
         ( j=m; j<dizi_boyutu; j=j+adım) için
            f[j]++;
      write (p1[ 1 ], mesaj, 1 );
      read (p1[ 0 ], mesaj, 1 );
   }
}
int  ana ( int argc, char *argv[])
{
   int i, j, len, ret, p1[ 2 ], adım= 0 , dizi_boyutu= 0 ;
   double *f, start_time, time1, min1=BÜYÜK, min2=BÜYÜK;
   char message, ch;
   short round ;
   pid_t p = 0 ;
   unsigned  long new_mask = 2 ;
   struct sched_param sp;

   len = sizeof (new_mask);
#ifdef MULTİPROCESSOR
   ret = sched_setaffinity (p, len, &new_mask);
   if(ret==- 1 ){
         perror ( " sched_setaffinity 1 " );
      exit( 1 );
```

```
}
   sp. sched_priority = sched_get_priority_max (SCHED_FIFO);
   ret= sched_setscheduler ( 0 , SCHED_FIFO, &sp);
   if (ret==- 1 ){
         perror ( " sched_setscheduler 1 " );
      exit ( 1 );
   }
# endif

   while ((ch = getopt (argc, argv, " s:n: " )) != EOF) {
      anahtar (ch) {
         case  ' n ' : /* dizideki çiftlerin sayısı */
            dizi_boyutu= atoi ( optarg );
           dizi_boyutu=dizi_boyutu/ sizeof ( double );
            break;
         case 's':
                     adım= atoi ( optarg );
                     adım=adım/ sizeof ( double );
            break ;
         default :
            fprintf (stderr, " Bilinmeyen seçenek karakteri. \n " );
                     showUsage ();
            exit ( 1 );
      }
}
   if (adım > dizi_boyutu){
      printf ( " Uyarı: adım dizi_boyutundan büyüktür. "
           " Sıralı erişim. \n " );
   }

   /* bir işlemde iki borunun yürütülmesini simüle edin */
   /* bir boru oluştur */
   if ( pipe (p1) < 0 ) {
      pererror ( " bir boru oluştur " );
      return - 1 ;
   }
   printf ( " zaman1 bağlam anahtarı olmadan: \t " );
```

```
fflush (stdout);
  /* N kez çalıştır ve minimum değeri al */
  for ( round = 0 ; round < N; round ++){
     flushCache ();
     f = ( double *) malloc (dizi_boyutu* sizeof ( double ));
     if (f== NULL ) {
        perror ( " malloc başarısız oldu " );
        exit ( 1 );
     }
          memset (( void *)f, 0x00 , array_size* sizeof ( double ));
     sleep ( 1 );
     start_time = gethrtime_x86 ();
     measureSingle (array_size, adım, p1, &message, f);
     time1 = gethrtime_x86 ()-start_time;
     time1 = time1/LOOP *MILLION;
     if (min1 > time1)
        min1 = time1;
     printf ( " %f \t " , zaman1);
     fflush (stdout);
     memdump (f, array_size* sizeof ( double ));
     free (f);
     sleep ( 1 );
  }
  printf ( " \n tek ölçü: dizi_boyutu = %lu , adım = %d , min süre1 = %.15f \n " ,
         array_size* sizeof ( double ), adım* sizeof ( double ), min1);
   return 0;
}
```

measureSwitch.c

```
/*
* AÇIKLAMA: Bir boru aracılığıyla iki işlem iletişimi.
* ÇIKIŞ: time2 = Dizi boyunca geçiş yükü + boru yükü
* + bağlam anahtarı ek yükü
 *
* time1 almak için measureSingle kullanın
* içerik değiştirme maliyeti = time2 - time1
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <pthread.h>
#include <sys/time.h>
#include <ctype.h>
#include <math.h>
#include <string.h>
#include <sched.h>
#include <errno.h>
#include <linux/unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "util.h"
void  showUsage ()
{
  fprintf ( stderr, " Kullanım: \n measureSwitch <seçenekler> \n \
    -n <sayı> üzerinde çalışılacak dizinin boyutu (bayt olarak). varsayılan 0. \n \
    -s <sayı> erişim adım boyutu (bayt cinsinden). varsayılan 0 \n " );
}
void measureSwitch1(register int array_size, register int stride,
            register int *p1, register int *p2, register char *msg,
            register double *f){

   register int i, j, m;
```

```
    for ( i=0; i<LOOP; i++) {
        read(p2[0], msg, 1);
        for ( m=0; m<stride; m++)
            for ( j=m; j<array_size; j=j+stride)
                f[j]++;
        write(p1[1], msg, 1);
    }
}
int main(int argc, char *argv[])
{
    int i, j, len, ret, p1[2], p2[2], stride=0, array_size=0;
    double *f, start_time, time1, min2=LARGE;
    char message, ch;
    short round;
    pid_t pid, p = 0;
    unsigned long new_mask = 2;
    struct sched_param sp;

    len = sizeof(new_mask);
#ifdef MULTIPROCESSOR
    ret = sched_setaffinity(p, len, &new_mask);
    if(ret==-1){
            perror("sched_setaffinity 1");
        exit(1);
    }
    sp.sched_priority = sched_get_priority_max(SCHED_FIFO);
    ret=sched_setscheduler(0, SCHED_FIFO, &sp);
    if(ret==-1){
            perror("sched_setscheduler 1");
        exit(1);
    }
#endif

    while ((ch = getopt(argc, argv, "s:n:")) != EOF) {
        switch (ch) {
            case 'n': /* number of doubles in the array */
                array_size=atoi(optarg);
```

```
        array_size=array_size/sizeof(double);
        break;
      case 's':
                stride=atoi(optarg);
                stride=stride/sizeof(double);
        break;
      default:
        fprintf(stderr, "Bilinmeyen seçenek karakteri.\n");
        showUsage();
        exit(1);
   }
}
if (stride > array_size){
   printf("Uyarı: adım dizi_boyutundan büyüktür. "
       "Sıralı erişim. \n");
}

/* create two pipes: p1[0], p2[0] for read; p1[1], p2[1] for write */
if (pipe (p1) < 0) {
   perror ("create pipe1");
   return -1;
}
if (pipe (p2) < 0) {
   perror ("create pipe2");
   return -1;
}
/* iki işlem arasında iletişim kurun*/
// fork
printf("time2 with context swith: \t");
fflush(stdout);
for(round=0; round<N; round++){

      flushCache();
   if ((pid = fork()) <0) {
      perror("fork");
      return -1;
   } else if (pid ==0) {
   // child process
```

```
        f = (double*) malloc(array_size*sizeof(double));
        if (f==NULL) {
           perror("malloc fails");
           exit (1);
        }
            memset((void *)f, 0x00, array_size*sizeof(double));
        measureSwitch1(array_size, stride, p1, p2, &message, f);
            sleep(1);
        memdump(f, sizeof(double)*array_size);
        free(f);
            exit(0);
      } else {
        // parent process
        f = (double*) malloc(array_size*sizeof(double));
        if (f==NULL) {
           perror("malloc fails");
           exit (1);
        }
            memset((void *)f, 0x00, array_size*sizeof(double));
            sleep(1);
        start_time = gethrtime_x86();
        measureSwitch2(array_size, stride, p1, p2, &message, f);
        time1 = gethrtime_x86()-start_time;
            time1 = time1/(2*LOOP)*MILLION;
        if(min2 > time1)
           min2 = time1;
        printf("%f\t", time1);
        fflush(stdout);
        waitpid(pid, NULL, 0);
        memdump(f, sizeof(double)*array_size);
        free(f);
      }
          sleep(1);
   }
   printf("\nmeasureSwitch: array_size = %lu, stride = %d, min time2 = %.15f\n",
          array_size*sizeof(double), stride*sizeof(double), min2);
   return 0;
}
```

Util.c

```
/*
        * AÇIKLAMA: Bağlam değiştirme maliyeti ölçümü için yardımcı
        program
        * x86 mimarisinde yüksek çözünürlüklü bir zamanlayıcı dahil
         */

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "util.h
/* başlangıçtan beri geçen süreyi (saniye olarak) al */
double gethrtime_x86(void)
{
   static double CPU_MHZ=0;
   if (CPU_MHZ==0) CPU_MHZ=getMHZ_x86();
   return (gethrcycle_x86()*0.000001)/CPU_MHZ;
}

/*başlangıçtan beri CPU döngü sayısını al */
hrtime_t gethrcycle_x86(void)
{
   unsigned int tmp[2];

   __asm__ ("rdtsc"
            : "=a" (tmp[1]), "=d" (tmp[0])
            : "c" (0x10) );

   return ( ((hrtime_t)tmp[0] << 32 | tmp[1]) );
}
/* Linux /proc dosya sisteminden mikrosaniye başına CPU döngü sayısını al
* hata durumunda <0 döndürür
 */
```

```
double getMHZ_x86(void)
{
   double mhz = -1;
   char line[1024], *s, search_str[] = "cpu MHz";
   FILE *fp;

   /* proc/cpuinfo'yu aç*/
   if ((fp = fopen("/proc/cpuinfo", "r")) == NULL)
           return -1;

   /*  MHz bilgisine ulaşana kadar tüm satırları yok sayın */
   while (fgets(line, 1024, fp) != NULL) {
           if (strstr(line, search_str) != NULL) {
              /*şu satıra kadar olan tüm karakterleri yoksay: */
              for (s = line; *s && (*s != ':'); ++s);
              /* get MHz number */
              if (*s && (sscanf(s+1, "%lf", &mhz) == 1))
                      break;
           }
   }

   if (fp!=NULL) fclose(fp);

   return mhz;
}
/* bir bellek parçasını /dev/null'a boşalt
 */
void memdump(double *m, int bytes)
{
   int fd;

   if ((fd=open("/dev/null",O_WRONLY)) == -1){
           perror("/dev/null open error");
      exit(-1);
   }
   if (write(fd, (void *)m, bytes) == -1){
           perror("/dev/null write error");
```

```
   exit(-1);
   }
   if (close(fd) != 0){
         perror("/dev/null close error");
      exit(-1);
   }
}
/* önbelleği temizle
 */
void flushCache()
{
   char *f;
   static char foo=0;
      f = (char*) malloc(FLUSHSIZE);
      if (f==NULL) {
         perror("malloc fails");
         exit (1);
      }
      memset((void *)f, foo++, FLUSHSIZE);
      free(f);
}
```

Util.h

```
/*
   * DOSYA: util.h
   * AÇIKLAMA: util.c için başlık dosyası
    */

#define N      3
#define LOOP    10000
#define MILLION 1000000
#define LARGE   100000000
#define FLUSHSIZE       4194304
#define MULTIPROCESSOR

typedef long long hrtime_t;
/* başlangıçtan beri geçen süreyi (saniye olarak) al */
double gethrtime_x86(void);
/* başlangıçtan beri CPU döngü sayısını al */
hrtime_t gethrcycle_x86(void);
/* Linux /proc dosya sisteminden mikrosaniye başına CPU döngü
sayısını al */
double getMHZ_x86(void);

/* dev/null'a bir yığın bellek boşaltın*/
void memdump(double *m, int bytes);

/* önbelleği temizle*/
void flushCache();
```

SYSTEM CALL (SİSTEM ÇAĞRI)MALİYETİ

Bu program oldukça basittir.

- rdtscp işlevi, RTDSCP yönergesini (64 bitlik döngü sayısını iki adet 32 bitlik kayda yükleyen bir işlemci yönergesi) çağırmak için yalnızca bir sarıcıdır. Bu fonksiyon zamanlamayı almak için kullanılır.

```
#include <iostream>

#include <unistd.h>

#include <sys/time.h>

uint64_t

rdtscp(void) {

  uint32_t eax, edx;



  __asm__ __volatile__("rdtscp" //! rdtscp instruction

            : "+a" (eax), "=d" (edx) //! output

            : //! input

            : "%ecx"); //! registers

  return (((uint64_t)edx << 32) | eax);

}
```

```
int main(void) {
  uint64_t before;
  uint64_t after;
  struct timeval t;
  for (unsigned int i = 0; i < 10; ++i) {
    before = rdtscp();
    gettimeofday(&t, NULL);
    after = rdtscp();
    std::cout << after - before  << std::endl;
    std::cout << t.tv_usec << std::endl;
  }

  return 0;
}
```