

## Mekanizma: Sınırlı Doğrudan Yürütme (Mechanism: Limited Direct Execution)

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 sanallaştırma sağlanır.

Bununla birlikte, bu tür sanallaştırma makinelerini oluşturmanın birkaç zorluğu vardır. Birincisi performans (*performance*): sisteme aşırı yük eklemeyi sanallaştırmayı nasıl uygulayabiliriz? İkincisi kontrol (*control*): CPU üzerinde kontrolü korurken süreçleri verimli bir şekilde nasıl çalıştırabiliriz? Kontrol (Control), 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. Kontrolü sürdürürken yüksek performans elde etmek, bir işletim sistemi oluşturmanın temel zorluklarından biridir.

### Önemli Nokta:

#### İŞLEMÇİ 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 Yürütme

### (Basic Technique: Limited Direct Execution)

Bir programın beklendiği kadar hızlı çalışmasını sağlamak için, işletim sistemi geliştiricilerinin **sınırlı doğrudan yürütme (limited direct execution)** dediğimiz bir teknik bulmaları şaşırtıcı değildir. Fikrin “Dorudan uygulama (direct execution)” kısmı basittir: programı doğrudan CPU üzerinde çalıştırmanız yeterlidir. Böylece, işletim sistemi çalışan bir programı başlatmak istediğinde, kendisi için bir süreç listesinde bir süreç girişi oluşturur, bunun için biraz bellek ayırır, program kodunu belleğe (diskten) yükler, giriş noktasını bulur (i.e., the `main()` rutin veya benzeri bir şey), atlar (jumps)

İşletim Sistemi (OS)	Program
İşlem listesi için girdi oluştur	
Program için bellek ayır	
Programı belleğe yükle	
argc/argv ile yığını (stack)	
ayarla	
Kayıtları (Reg.) temizle	
call main() uygula	Run main()
	Main'den return uygula
İşlem listesinden hafızayı boşalt	
İşlem listesinden kaldır	

### Şekil 6.1: Doğrudan Yürütme Protokolü (Sınırsız)

#### Direct Execution Protocol (Without Limits)

Şekil 6.1, bu temel doğrudan yürütme protokolünü göstermektedir (henüz herhangi bir sınırlama olmaksızın), normal bir çağrı kullanarak ve programın main () atlamak ve daha sonra çekirdeğe geri dönmek için geri dönün.

Kulağa basit geliyor, değil mi? Ancak bu yaklaşım, CPU'yu sanallaştırma arayışımızda birkaç soruna yol açıyor. İlki basit: Eğer sadece bir program çalıştırsak, işletim sistemi, programı verimli bir şekilde çalıştırırken, programın yapmasını istemediğimiz hiçbir şeyi yapmadığından nasıl emin olabilir? İkinci: biz bir processi çalıştırırken işletim sistemi onu nasıl durdurur ve başka bir process'e geçer, böylece CPU'yu sanallaştırmak için ihtiyaç duyduğumuz **zaman paylaşımını (time sharing)** uyguluyor muyuz?

Aşağıdaki bu soruları cevaplarken, 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 (Restricted Operations)

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

**Önemli Nokta: KISITLANMIŞ İŞLEMLER NASIL GERÇEKLEŞTİRİLİR** Bir işlem, G/Ç ve diğer bazı kısıtlanmış işlemleri gerçekleştirebilmelidir. ancak sürece sistem üzerinde tam kontrol vermeden. İşletim sistemi ve donanım bunu yapmak için nasıl birlikte çalışabilir?

### KENAR: SİSTEM ÇAĞRILARI NEDEN PROSEDÜR ÇAĞRILARI GİBİ GÖRÜNÜYOR

Bir sistem çağrısının neden çağrıldığını merak edebilirsiniz. `open()` ve `read()` gibi, tam olarak C'deki tipik bir prosedür çağrısına benziyor; yani, sadece bir prosedür çağrısı gibi görünüyorsa, sistem bunun bir sistem çağrısı olduğunu nasıl biliyor ve tüm doğru şeyleri yapıyor? Basit sebep: bu bir prosedür çağrısı, ancak bu prosedür çağrısının içinde ünlü tuzak talimatı gizlidir. Daha spesifik olarak, `open()` metodunu çağırdağında (Örnek olarak), C kitaplığına bir prosedür çağrısı yürütüyorsunuz. Burada, `open()` veya sağlanan diğer sistem çağrılarından herhangi biri için, kitaplık, argümanları iyi bilinen konumlarda `open()`'a koymak için çekirdek üzerinde anlaşmaya varılan bir çağrı kuralını kullanır (e.g., on the stack, or in specific registers), sistem çağrı numarasını da iyi bilinen bir konuma yerleştirir (tekrar yığına veya bir kayda), ve daha sonra yukarıda belirtilen tuzak talimatını yürütür. Tuzak paketini açtıktan sonra kitaplıktaki kod, dönüş değerlerini açar ve kontrolü sistem çağrısını yapan programa geri verir. Böylece C kütüphanesinin sistem çağrıları yapan kısımları, montajda el ile kodlanmış, argümanları işlemek ve değerleri doğru bir şekilde döndürmek için kuralları dikkatlice takip etmeleri gerektiğinden, ayrıca donanıma özgü tuzak talimatını yürütür. 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 denetleyen bir dosya sistemi oluşturmak istiyorsak, herhangi bir kullanıcının diske G/Ç 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 G/Ç isteklerini yayınlamaz; bunu yapmak, işlemcinin bir istisna oluşturmaya neden olur; işletim sistemi daha sonra muhtemelen süreci öldürür.

Kullanıcı modunun aksine **çekirdek modudur (kernel mode)**, işletim sisteminin (veya çekirdeğin (kernel)) içinde çalışır. Bu kipte, çalışan kod, G/Ç isteklerini yayınlama ve her türlü kısıtlanmış talimatı yürütme gibi ayrıcalıklı işlemler dahil, istediğini yapabilir. Ancak yine de bir zorlukla karşı karşıyayız: Bir kullanıcı işlemi, bir tür ayrıcalıklı işlem gerçekleştirmek istediğinde ne yapmalıdır, diskten okuma gibi? Bunu etkinleştirmek için, neredeyse tüm modern donanımlar, kullanıcı programlarının bir **sistem çağrısı (System call)** gerçekleştirmesini sağlar. Atlas [K+61,L78] gibi eski makinelerde öncülük edilen sistem çağrıları, çekirdeğin, dosya sistemine erişim, süreçler oluşturma ve yok etme gibi belirli temel işlevsellik parçalarını kullanıcı programlarına dikkatlice göstermesine izin verir, diğer süreçlerle iletişim kurmak ve daha fazlası tahsis etmektedir.

## 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 (Kernal mode)**, işletim sisteminin makinenin tüm kaynaklarına erişimi vardır.

Çekirdeğe **tuzak (trap)** girişi ve **trap**'dan kullanıcı modu programlarına **geri dönüş (return)** için özel talimatlar da sağlanır, işletim sisteminin donanıma **trap tablosunun** bellekte nerede olduğunu söylemesine izin veren talimatlar.

Çoğu işletim sistemi yüzlerce **çağrı (calls)** sağlar (ayrıntılar için POSIX standardına bakın [P10]); erken Unix sistemleri, yaklaşık yirmi **çağrı**dan oluşan daha özlü bir alt küme ortaya çıkardı.

Bir sistem çağrısını yürütmek için, bir programın özel bir **tuzak (trap)** talimatı 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 veriliyorsa), ve böylece arama işlemi için gerekli çalışmaları yapar. Bittiğinde, OS özel bir **tuzaktan dönüş (return from trap)** talimatı çağırır, bu, beklediğiniz gibi, aynı anda ayrıcalık seviyesini tekrar kullanıcı moduna düşürürken çağırana kullanıcı programına geri döner.

Bir tuzak yürütülürken donanımın biraz dikkatli olması gerekir, çünkü işletim sistemi **tuzaktan dönüş (return from trap)** talimatını verdiğinde doğru şekilde geri dönebilmek için arayanın kayıtlarından yeterince tasarruf ettiğinden emin olmalıdır. Örneğin, x86'da işlemci, program sayacını, **bayrakları (flags)** ve diğer birkaç **kaydı (reg.)** işlem başına bir **çekirdek yığınının (kernel stack)** gönderir; **tuzaktan dönüş (return from trap)**, bu değerleri **yığından (stack)** çıkaracak ve **kullanıcı modu (user mode)** programının yürütülmesine devam edecektir (ayrıntılar için Intel sistem kılavuzlarına [I11] bakın). Diğer donanım sistemleri farklı kuralları kullanır, ancak temel kavramlar platformlar arasında benzerdir.

Bu tartışmanın dışında kalan önemli bir ayrıntı var: **Tuzak (trap)** işletim sisteminde hangi kodu çalıştıracağını nasıl biliyor? Açıkçası, çağırma işlemi atlanacak bir adres belirleyemez (bir prosedür çağrısı yaparken yaptığınız gibi); bunu yapmak, programların **çekirdeğe (kernal)** herhangi bir yere atlamasına izin verir ki bu açıkça Çok Kötü Bir Fikirdir. Bu nedenle çekirdek, bir tuzakta hangi kodun yürütüleceğini dikkatlice kontrol etmelidir.

Çekirdek bunu önyükleme sırasında bir **tuzak (trap) tablosu** kurarak yapar. Makine açıldığında, bunu ayrıcalıklı (kernal) moda yapar, ve böylece makine donanımını gerektiği gibi yapılandırmak ücretsizdir. İşletim 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.

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

OS @ boot (kernel mode)	Hardware	
initialize trap table	remember address of... syscall handler	
OS @ run (kernel mode)	Hardware	Program (user mode)
işlem listesi için girdi oluştur Program için bellek ayır Programı belleğe yükle argv ile kullanıcı yığınının ayarla Çekirdek yığınının reg/PC ile doldur <b>tuzaktan dön (return from trap)</b>	restore regs (from kernel stack) move to user mode jump to main	Run main() ... Call system call <b>trap</b> into OS
Handle trap Do work of syscall <b>return-from-trap</b>	save regs (to kernel stack) move to kernel mode jump to trap handler	
	restore regs (from kernel stack) move to user mode jump to PC after trap	... return from main <b>trap</b> (via <code>exit()</code> )
Free memory of process Remove from process list		

Şekil 6.2: Sınırlı Doğrudan Yürütme Protokolü

(Limited Direct Execution Protocol)

Bu **tuzak işleyicilerin (trap handlers)** adresleri, genellikle özel talimatlarla beraberdir. Donanım bilgilendirildikten sonra, makine yeniden başlatılana kadar bu işleyicilerin konumunu hatırlar ve böylece donanım, sistem çağrılarını ve diğer istisnai olaylar gerçekleştiğinde ne yapacağını (yani hangi koda atlayacağını) bilir.

### İPUCU: GÜVENLİ SİSTEMLERDE KULLANICI GİRİŞLERİNE KARŞI DİKKATLİ OLUN

Sistem çağrılarını sırasında işletim sistemini korumak için büyük çaba sarf etmemize rağmen (bir donanım yakalama mekanizması ekleyerek ve işletim sistemine yapılan tüm çağrılarının bu mekanizma üzerinden yönlendirilmesini sağlayarak), **güvenli (secure)** 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.

Genel olarak, güvenli bir sistem, kullanıcı girişlerine büyük bir şüpheyle yaklaşmalıdır. Bunu yapmamak şüphesiz yazılımların kolayca hacklenmesine, dünyanın güvensiz ve korkutucu bir yer olduğuna dair umutsuz bir duyguya ve fazlasıyla güvenen işletim sistemi geliştiricisi için iş güvenliğini kaybetmesine yol açacaktır.

Kesin sistem çağrısını belirtmek için, genellikle her sistem çağrısına bir **sistem çağrı numarası (system call number)** 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 (protection)** 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 **ayrıcılık bir işlemdir (privileged operation)**. 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 kurabilirseniz, bir sisteme ne gibi korkunç şeyler yapabilirsiniz? 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 yönergeler koyu renkle vurgulanmıştır).

İkincisinde (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 işlemiz biter.

### 6.3 Problem #2: Süreçler Arasında Geçiş (Switching Between Processes)

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 (cooperative)** 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 ütopya 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. Örneğin, bir uygulama sıfıra bölerse veya erişememesi gereken belleğe erişmeye çalışırsa, işletim sisteminde bir tuzak (trap) oluşturur.

İşletim sistemi daha sonra CPU'nun kontrolünü tekrar ele geçirecek (ve muhtemelen rahatsız edici süreci sonlandıracaktır).

Böylece, işbirlikçi bir zamanlama sisteminde, işletim sistemi, bir sistem çağrısını veya bir tür yasa dışı işlemin gerçekleşmesini bekleyerek CPU'nun kontrolünü yeniden kazanır. Şunu da düşünebilirsiniz: Bu pasif yaklaşım idealden daha az değil mi? Örneğin, bir işlem (kötü niyetli veya yalnızca hatalarla dolu) sonsuz bir döngüde sona ererse ve asla bir sistem çağrısı yapmazsa ne olur? İşletim sistemi o zaman ne yapabilir?

### İşbirlikçi Olmayan Bir Yaklaşım: İşletim Sistemi Kontrolü Ele Geçiriyor

Donanımdan bazı ek yardımlar olmadan, bir işlem 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 başvurmaktır: **makineyi yeniden başlat (reboot the machine)**. 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 ve yıllar önce bilgisayar sistemleri kuran birkaç kişi tarafından keşfedildiği ortaya çıktı: **zamanlayıcı kesintisi (timer interrupt) [M+63]**. Bir zamanlayıcı cihazı, her milisaniyede bir kesinti oluşturacak şekilde programlanabilir; kesme yükseltildiğinde, o anda çalışan işlem durdurulur, ve işletim sisteminde önceden yapılandırılmış bir kesme işleyicisi çalışır. Bu noktada, işletim sistemi CPU'nun kontrolünü yeniden ele geçirdi ve böylece 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 (kötülük) 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?



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 yığınının) 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 (scheduler)** olarak bilinen bir bölümü tarafından verilir; sonraki birkaç bölümde çizelgeleme politikalarını ayrıntılı olarak tartışacağız.

Değiştirme kararı verilirse, işletim sistemi daha sonra **bağlam anahtarı (context switch)** olarak adlandırdığımız düşük seviyeli bir kod parçasını yürütür. Bağlam anahtarı kavramsal olarak basittir: işletim sisteminin tek yapması gereken, şu anda yürütülen işlem için birkaç kayıt değeri kaydetmek (örneğin, çekirdek yığınının) ve yakında yürütülecek 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.

Hali hazırda 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 yığınının 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 ile 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 yönetim 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. Süreç B (süreç yapısı girişinden), ve ardından, özellikle B'nin çekirdek yığınına (A'nın değil) kullanmak için yığın işaretçisini değiştirerek **bağlamları değiştirir (switches contexts)**. 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 yığınına 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 açık bir şekilde kaydedilir, ancak bu kez sürecin süreç yapısındaki belleğe kaydedilir. İkinci eylem, sistemi A'dan çekirdeğe yeni sıkışmış gibi çalışmaktan B'den çekirdeğe yeni sıkışmış gibi çalıştırır.

Böyle bir anahtar 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
		...
	<b>timer interrupt</b> save regs(A) $\rightarrow$ k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call <code>switch()</code> routine save regs(A) $\rightarrow$ proc.t(A) restore regs(B) $\leftarrow$ proc.t(B) switch to k-stack(B) <b>return-from-trap (into B)</b>	restore regs(B) $\leftarrow$ k-stack(B) move to user mode jump to B's PC	
		Process B
		...

Figure 6.3: Limited Direct Execution Protocol (Timer Interrupt)

#### 6.4 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.

```

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

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.

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 işlemcilerde 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.

## 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, işletim sistemi yardımcı olmadan işlemin yapabileceklerini sınırlandırmak için donanımı kurduğunuzdan emin olun.

**KENAR: 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 **Imbench** [MS96] adlı bir araç ve ilgili olabilecek birkaç başka performans ölçüsü vardır.

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 yoğunudur 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, aranızda çocuğu olan veya en azından çocukları duyanlarınız, **bir odayı bebek için hazırlama (baby proofing a room)** konseptine aşina olabilirsiniz: tehlikeli maddeler içeren ve elektrik prizlerini kapatan kilitli dolaplar. 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 cevaplama gereken soru bu ve dolayısıyla çalışmamızın bir sonraki konusu.

**KENAR: ANAHTAR CPU SANALLATMA ŞARTLARI (MEKANİZMALAR)**

- CPU en az iki yürütme modunu desteklemelidir: kısıtlı **kullanıcı modu (user mode)** ve ayrıcalıklı (sınırsız) **çekirdek modu (kernel mode)**.
- Tipik kullanıcı uygulamaları, kullanıcı modunda çalışır ve bir **sistem çağrısı (system call)** kullanır. İşletim sistemi hizmetleri istemek için çekirdeğe **tuzak (trap)** kurmak.
- Tuzak talimatı, kayıt durumunu dikkatli bir şekilde kaydeder, donanım durumunu çekirdek moduna değiştirir ve OS'de önceden belirlenmiş bir hedefe atlar: **tuzak tablosu (trap table)**.
- İşletim sistemi bir sistem çağrısına hizmet vermeyi bitirdiğinde, ayrıcalığı azaltan ve denetimi işletim sistemine sıçrayan tuzaktan sonra yönergeye geri döndüren başka bir özel **tuzaktan dönüş (return from trap)** 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 çalışıyorsa, işletim sisteminin kullanıcı programının sonsuza kadar çalışmamasını sağlamak için donanım mekanizmalarını, yani **zamanlayıcı kesintisini (timer interrupt)** kullanması gerekir. 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 of the channel 17 clock routine is to decide whether to remove the current user from core and if so to decide which user program to swap in as he goes out.”*
- [MS96] “Imbench: 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 Imbench and give it a try.*
- [M11] “Mac OS 9” by Apple Computer, Inc.. January 2011. [http://en.wikipedia.org/wiki/Mac\\_OS\\_9](http://en.wikipedia.org/wiki/Mac_OS_9). *You can probably even find an OS 9 emulator out there if you want to; check it out, it’s a fun little Mac!*
- [O90] “Why Aren’t Operating Systems Getting Faster as Fast as Hardware?” by J. Ousterhout. USENIX Summer Conference, June 1990. *A classic paper on the nature of operating system performance.*
- [P10] “The Single UNIX Specification, Version 3” by The Open Group, May 2010. Available: <http://www.unix.org/version3/>. *This is hard and painful to read, so probably avoid it if you can. Like, unless someone is paying you to read it. Or, you’re just so curious you can’t help it!*
- [S07] “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)” by Hovav Shacham. CCS ’07, October 2007. *One of those awesome, mind-blowing ideas that you’ll see in research from time to time. The author shows that if you can jump into code arbitrarily, you can essentially stitch together any code sequence you like (given a large code base); read the paper for the details. The technique makes it even harder to defend against malicious attacks, alas.*

## Ödev (Ölçme)

### KENAR: 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, `lbench` bir bağlam değişikliğinin maliyetine ilişkin iyi bir tahminde bulunabilir. Boruları veya belki de UNIX soketleri gibi başka bir iletişim mekanizmasını kullanarak benzer bir şeyi burada 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.



### Çözüm :

a-) System Çağırısı (**System call**) Maliyeti :

**gettimeofday()** işlevi, sistemin saat zamanını alır. Geçerli zaman, 1 Ocak 1970 (Unix Epoch) 00:00:00'dan bu yana geçen saniye ve mikrosaniye cinsinden ifade edilir. Başarı durumunda, **gettimeofday()** işlevi 0 döndürür, başarısızlık durumunda işlev -1 döndürür.

### Basit gettimeofday() ve Yazdırılması

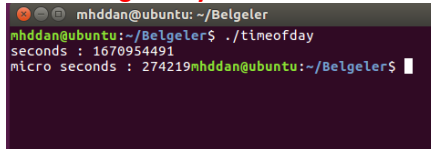
```
#include <sys/time.h>
#include <stdio.h>

int main() {
    struct timeval current_time;
    gettimeofday(&current_time, NULL);
    printf("seconds : %ld\nmicro seconds : %ld",
        current_time.tv_sec, current_time.tv_usec);
    return 0;
}
```

Text dosyası oluştururuz ve içine bu kodu yazarız, daha sonra o Text dosyasının bulunduğu konumdan **“Uçbirim (Terminal)”** açarız, sonra içine **gcc derleyicisini (Compile)** kullanarak derleme işlemi tamamlıyoruz (**gcc -o TextDosyaismi TextDosyaismi.c**), bu işlemi tamamladıktan sonra yeni dosyamız oluşur ve **gettimeofday()** işlevini kullanabiliriz artık,

**./gettimeofday()** yazarak saniye ve mikro saniye cinsinden biz sonucu verir.

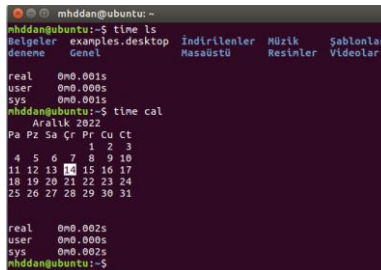
### Aşağıdaki resim sonucu gösteriyor :



```
mhddan@ubuntu: ~/Belgeler
mhddan@ubuntu:~/Belgeler$ ./timeofday
seconds : 1670954491
micro seconds : 274219mhddan@ubuntu:~/Belgeler$
```

Aynı zamanda çok basit bir kodla **“time”** emrini kullanarak bir işlevin ne kadar zamanda yapıldığını gösteren sonuçla karşılaşırız.

Örnek : **“time ls”** VEYA **“time cal”** gibi yapmak istediğimiz işlemden önce **“time”** yazarak sonuç elde ederiz



```
mhddan@ubuntu: ~
mhddan@ubuntu:~$ time ls
Belgeler  examples.desktop  İndirilenler  Müzik  Şablonlar
deneme    Gencl              Masaüstü      Resimler  Videolar
real    0m0.001s
user    0m0.000s
sys     0m0.001s
mhddan@ubuntu:~$ time cal
    Aralık 2022
Su  Pz  Sa  Çr  Pr  Cu  Ct
            1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

real    0m0.002s
user    0m0.000s
sys     0m0.002s
mhddan@ubuntu:~$
```

Yukarıdaki resimde göreceğimiz üzere bize “real, user ve sys” açılarından geçen süreyi verir.

Başka bir örnekl daha fazla detay ihtiyacımız varsa eğer “/usr/bin/time -v” kodunu kullanabiliriz

Bu kodu yapmak istediğimiz işlemden öncesine yazarız ve bize birçok detay verir :

```

phd@ubuntu:~$ /usr/bin/time -v free
              total        used        free      shared  buff/cache   available
Mem: 4015896    1286532    1482364       7188      1327188    2344992
Swap:  998396           0     998396

Command being timed: "free"
User time (seconds): 0.00
System time (seconds): 0.00
Percent of CPU this job got: 15%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.01
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 3124
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 146
Voluntary context switches: 4
Involuntary context switches: 0
Swaps: 0
File system inputs: 56
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0

```

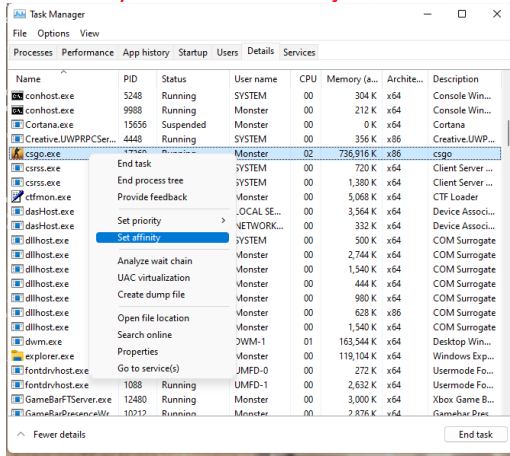
Resimdeki örnekte üstünde durmak istediğim bir nokta var “Voluntary context switches: 4” bunun anlamı yaptığımız işlemin CPU’yu kaç defa ziyaret ettiği bilgisini verir.

“free” işlemi göreceğimiz üzere tamamlanması için 4 defa CPU’yu ziyaret ettiği görülür yani bu işlemin tamamlanması için 4 zaman dilimi almıştır.

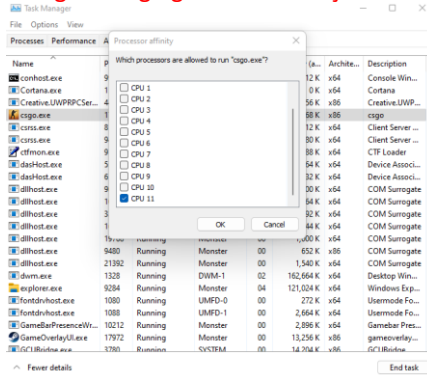
### b-) Bağlam anahtarı (context switch)

bu deneyde aynı CPU’de 2 tane programı çalıştıracamız ve CPU kullanım grafiklerdeki değişimi göreceğiz

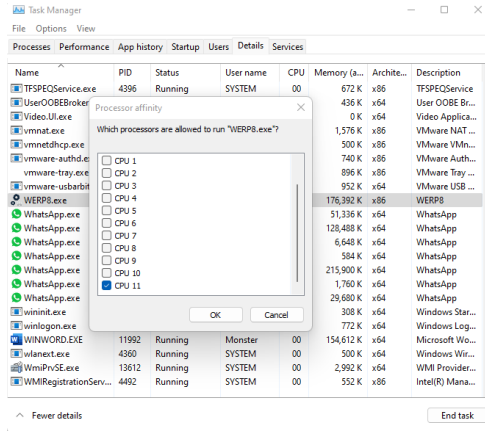
İlk olarak Görev çubuğunu açıyoruz ve işlemler kısmına geliriz, istediğimiz uygulamaya sağ tıklayıp detaylar butonuna tıklarız, daha sonra tekrar sağ tıklayıp aşağıdaki resimde görüldüğü gibi “Set affinity” butonuna tıklayarak bir tane CPU işaretleriz



Aşağıdaki resimde görüldüğü gibi CPU 11'i seçtim :



Daha sonra başka bir uygulamayı seçip aynı işlemi tekrarlarız (Aşağıdaki resimde görüldüğü gibi) :



Aşağıdaki resimde görüldüğü gibi CPU 11 tamamen yüklendi ve diğer CPU'lardan çok daha fazla aktivite gösteriliyor :

