# 15

# Mekanizma: Adres Çevirisi(Address Translation)

CPU'nun sanallaştırılmasını geliştirirken, **sınırlı doğrudan yürütme(limited direct execution)** (veya **LDE**) olarak bilinen genel bir mekanizmaya odaklandık. LDE'nin arkasındaki fikir basittir: çoğunlukla, programın doğrudan donanım üzerinde çalışmasına izin verin; ancak, zaman içindeki belirli kilit noktalarda (bir süreç bir sistem çağrısı yayınladığında veya bir zamanlayıcı kesintisi meydana geldiğinde), işletim sisteminin devreye girmesini ve "doğru" şeyin gerçekleşmesini sağlayacak şekilde ayarlayın. Böylece işletim sistemi, küçük bir donanım desteğiyle, *verimli* bir sanallaştırma sağlamak için çalışan programın yolundan çekilmek için elinden geleni yapar; ancak, zamanın bu kritik noktalarında **araya gir**erek, işletim sistemi donanım üzerindeki *kontrolünü* sürdürmesini sağlar. Verimlilik ve kontrol birlikte her modern işletim sisteminin ana hedeflerinden ikisidir.

Belleğin sanallaştırılmasında da benzer bir strateji izleyeceğiz ve istenen sanallaştırmayı sağlarken hem verimlilik hem de kontrol elde edeceğiz. Verimlilik, başlangıçta oldukça ilkel olacak (örneğin sadece birkaç yazmaç) ancak daha sonra oldukça karmaşık hale gelecek (örneğin TLB'ler, sayfa tablosu desteği vb.) donanım desteğinden yararlanmamızı gerektirir. Kontrol, işletim sisteminin hiçbir uygulamanın kendi belleği dışında herhangi bir belleğe erişmesine izin vermemesini sağlaması anlamına gelir; bu nedenle, uygulamaları birbirinden ve işletim sistemini uygulamalardan korumak için burada da donanımın yardımına ihtiyacımız olacaktır. Son olarak, *esneklik* açısından VM sisteminden biraz daha fazlasına ihtiyacımız olacak; özellikle, programların adres alanlarını istedikleri şekilde kullanabilmelerini ve böylece sistemin programlanmasını kolaylaştırmak istiyoruz. Ve böylece rafine edilmiş en önemli noktaya ulaşıyoruz:

THE CRUX:
BELLEK NASIL VERIMLI VE ESNEK BIR ŞEKILDE SANALLAŞTIRILIR

Belleğin verimli bir şekilde sanallaştırılmasını nasıl sağlayabiliriz? Uygulamaların ihtiyaç duyduğu esnekliği nasıl sağlayabiliriz? Bir uygulamanın hangi bellek konumlarına erişebileceğini nasıl kontrol altında tutarız ve böylece uygulama bellek erişimlerinin uygun şekilde kısıtlanmasını nasıl sağlarız? Tüm bunları nasıl verimli bir şekilde yapabiliriz?

Kullanacağımız genel teknik, sınırlı doğrudan yürütme genel yaklaşımımıza bir ek olarak düşünebileceğiniz, **donanım tabanlı adres çevirisi(hardware-based address translation)** veya kısaca **adres çevirisi olarak(address trans- lation)** adlandırılan bir şeydir. Adres çevirisi ile donanım her bir bellek erişimini (örneğin bir komut getirme, yükleme ya da saklama) dönüştürerek komut tarafından sağlanan **gerçek** adresi istenen bilginin gerçekte bulunduğu **fiziksel** adrese çevirir. Böylece, her bir bellek referansında, uygulama belleği referanslarını bellekteki gerçek konumlarına yönlendirmek için donanım tarafından bir adres çevirisi gerçekleştirilir.

Elbette, donanım tek başına belleği sanallaştıramaz, çünkü bunu verimli bir şekilde yapmak için sadece düşük seviyeli mekanizma sağlar. İşletim sistemi, donanımı doğru aktarımların gerçekleşeceği şekilde ayarlamak için kilit noktalarda devreye girmelidir; bu nedenle **belleği yönetmeli**, hangi konumların boş, hangilerinin kullanımda olduğunu takip etmeli ve belleğin nasıl kullanıldığını kontrol altında tutmak için mantıklı bir şekilde müdahale etmelidir.

Bir kez daha tüm bu çalışmaların amacı güzel bir **illüzyon** yaratmaktır: programın kendi kod ve verilerinin bulunduğu kendi özel belleği vardır. Bu sanal gerçekliğin ardında çirkin bir fiziksel gerçek yatmaktadır: CPU (ya da CPU'lar) bir programı çalıştırmakla diğerini çalıştırmak arasında geçiş yaparken aslında birçok program aynı anda belleği paylaşmaktadır. Sanallaştırma sayesinde işletim sistemi (donanımın yardımıyla) çirkin makine gerçekliğini kullanışlı, güçlü ve kullanımı kolay bir soyutlamaya dönüştürür.

## 15.1 Varsayımlar

Belleği sanallaştırmaya yönelik ilk girişimlerimiz çok basit, neredeyse gülünecek kadar basit olacaktır. Devam edin, istediğiniz kadar gülün; çok yakında TLB'lerin, çok seviyeli sayfa tablolarının ve diğer teknik harikaların iç yüzünü anlamaya çalıştığınızda size gülen işletim sistemi olacak. İşletim sisteminin size gülmesi fikri hoşunuza gitmiyor mu? O zaman şansınız yok demektir; işletim sistemi böyle çalışır.

Özellikle, şimdilik kullanıcının adres alanının fiziksel belleğe *bitişik olarak* yerleştirilmesi gerektiğini varsayacağız. Ayrıca, basit olması için, adres alanının boyutunun çok büyük olmadığını, özellikle de *fiziksel belleğin boyutundan daha küçük* olduğunu varsayacağız. Son olarak, her adres alanının tam olarak *aynı boyutta olduğunu* da varsayacağız. Bu varsayımlar gerçekçi gelmiyorsa endişelenmeyin; ilerledikçe bunları gevşeteceğiz, böylece belleğin gerçekçi bir sanallaştırmasını elde edeceğiz.

## 15.2 Bir Örnek

Adres aktarımını uygulamak için ne yapmamız gerektiğini ve neden böyle bir mekanizmaya ihtiyaç duyduğumuzu daha iyi anlamak için basit bir örneğe bakalım. Adres uzayı Şekil 15.1'de gösterildiği gibi olan bir süreç olduğunu düşünün. Burada inceleyeceğimiz şey, bellekten bir değer yükleyen, değeri üç artıran ve ardından değeri belleğe geri depolayan kısa bir kod dizisidir. Bu kodun C dilindeki gösteriminin aşağıdaki gibi olabileceğini hayal edebilirsiniz:

İPUCU: ARAYA GIRME GÜÇLÜDÜR

Araya girme, bilgisayar sistemlerinde sıklıkla kullanılan genel ve güçlü bir tekniktir. Belleği sanallaştırırken, donanım her bellek erişimine müdahale eder ve işlem tarafından verilen her sanal adresi, istenen bilginin gerçekte saklandığı fiziksel bir adrese çevirir. Bununla birlikte, genel müdahale tekniği çok daha geniş bir şekilde uygulanabilir; aslında, yeni işlevsellik eklemek veya sistemin başka bir yönünü geliştirmek için neredeyse iyi tanımlanmış herhangi bir arayüze müdahale edilebilir. Böyle bir yaklaşımın olağan faydalarından biri **şeffaflıktır(transparency)**; müdahale genellikle istemcinin ara yüzünü değiştirmeden yapılır, dolayısıyla söz konusu istemcide hiçbir değişiklik yapılmasını gerektirmez.

```
void func() {
    int x = 3000; // teşekkürler, Perry.
    x = x + 3;    // ilgilendiğimiz kod satırı
    ...
```

Derleyici bu kod satırını assembly'ye dönüştürür, bu da aşağıdaki gibi görünebilir (x86 assembly'de). Sökmek için Linux üzerinde `objdump` veya Mac üzerinde `otool` kullanın:

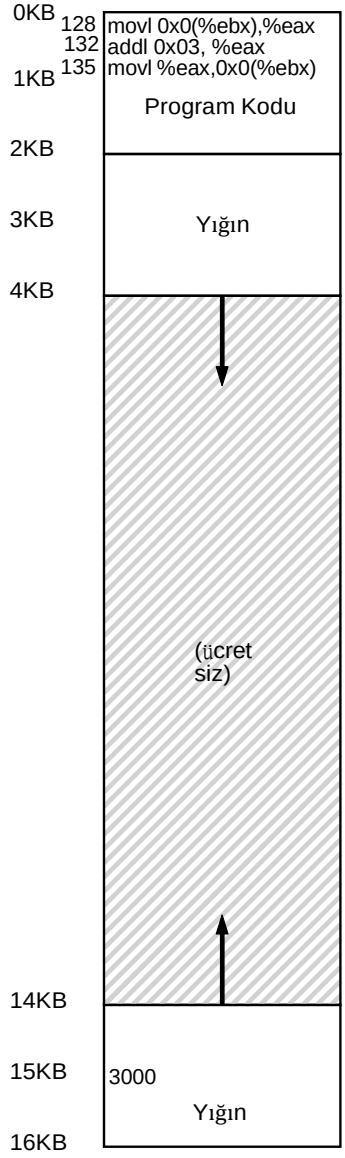
```
128: movl 0x0(%ebx), %eax     ;eax'a 0+ebx yükle 132: addl
$0x03                         ;eax yazmacına 3
ekle135: movl %eax, 0x0(%ebx) ;eax'ı mem'e geri
                              depola
```

Bu kod parçacığı oldukça basittir; `x` adresinin `ebx` yazmacına yerleştirildiğini varsayar ve ardından bu adresteki değeri `movl` yapısını ("uzun sözcük" taşıma için) kullanarak genel amaçlı `eax` yazmacına yükler. Bir sonraki komut `eax`'e 3 ekler ve son komut `eax`'deki değeri aynı konumda belleğe geri saklar.

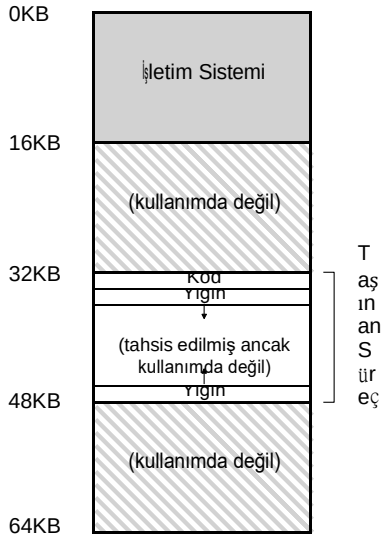
Şekil 15.1'de (sayfa 4), hem kodun hem de verilerin sürecin adres alanında nasıl düzenlendiğini gözlemleyin; üç komutlu kod dizisi 128 adresinde (üste yakın kod bölümünde) ve `x` değişkeninin değeri 15 KB adresinde (alta yakın yığında) bulunur. Şekilde, `x`'in başlangıç değeri, yığındaki konumunda gösterildiği gibi 3000'dir.

Bu talimatlar çalıştığında, süreç açısından bakıldığında, aşağıdaki bellek erişimleri gerçekleşir.

- Adres 128'de komut getirme
- Bu komutu çalıştırın (15 KB adresinden yükle)
- Adres 132'deki getirme komutu
- Bu komutu çalıştırın (bellek referansı yok)
- 135'inci adresteki komutu getirin
- Bu komutu çalıştırın (15 KB adresine saklayın)




Şekil 15.1: **Bir Süreç ve Adres Alanı**




Şekil 15.2: **Yeri Değiştirilmiş Tek Bir Süreç ile Fiziksel Bellek**

Programın bakış açısına göre, **adres alanı** 0 adresinden başlar ve en fazla 16 KB'a kadar büyür; oluşturduğu tüm bellek referansları bu sınırlar içinde olmalıdır. Ancak, belleği sanallaştırmak için, işletim sistemi süreci fiziksel bellekte başka bir yere yerleştirmek ister, ille de 0 adresine değil. Dolayısıyla şöyle bir sorunla karşı karşıyayız: Bu süreci, sürece **şeffaf** olacak şekilde bellekte nasıl **yeniden konumlandırabiliriz?** Gerçekte adres alanı başka bir fiziksel adreste bulunurken, 0'dan başlayan sanal bir adres alanı yanılsamasını nasıl sağlayabiliriz?

Bu sürecin adres alanı belleğe yerleştirildikten sonra fiziksel belleğin nasıl görünebileceğine dair bir örnek Şekil 15.2'de bulunmaktadır. Şekilde, işletim sisteminin fiziksel belleğin ilk yuvasını kendisi için kullandığını ve yukarıdaki örnekteki süreci fiziksel bellek adresi 32 KB'den başlayan yuvaya yerleştirdiğini görebilirsiniz. Diğer iki yuva boştur (16 KB-32 KB ve 48 KB-64 KB).

## 15.3 Dinamik (Donanım Tabanlı) Yer Değiştirme

Donanım tabanlı adres çevirisini anlamak için, ilk olarak ilk enkarnasyonunu tartışacağız. 1950'lerin sonlarındaki ilk zaman paylaşımlı makinelerde tanıtılan, **taban ve sınırlar** olarak adlandırılan basit bir fikirdir; teknik aynı zamanda **dinamik yer** değiştirme olarak da adlandırılır; her iki terimi de birbirinin yerine kullanacağız [SS74].

Özellikle, her CPU'da iki donanım kaydına ihtiyacımız olacak: birine **taban** kaydı, diğerine **sınırlar** (bazen sınır kaydı olarak da adlandırılır) denir. Bu base-and-bounds çifti, her bir CPU'ya

BIR KENARA: YAZILIM TABANLI YER DEĞIŞTIRME

İlk zamanlarda, donanım desteği ortaya çıkmadan önce, bazı sistemler yalnızca yazılım yöntemleriyle kaba bir yer değiştirme biçimi oluşturmuştur. Temel teknik, **yükleyici** olarak bilinen bir yazılım parçasının çalıştırılmak üzere olan bir yürütülebilir dosyayı aldığı ve adreslerini fiziksel bellekte istenen ofsete yeniden yazdığı **statik yer değiştirme olarak** adlandırılır.

Örneğin, bir komut 1000 adresinden bir reg- ister'e yükleniyorsa (örneğin, `movl 1000, %eax`) ve programın adres alanı 3000 adresinden başlayarak yükleniyorsa (programın düşündüğü gibi 0 değil), yükleyici komutu her adresi 3000 ile kaydıracak şekilde yeniden yazacaktır (örneğin, `movl 4000, %eax`). Bu şekilde, sürecin adres alanının basit bir statik yer değiştirmesi sağlanır.

Ancak statik yer değiştirmenin çok sayıda sorunu vardır. Birincisi ve en önemlisi, süreçler kötü adresler üretebileceğinden ve böylece diğer süreçlerin ve hatta işletim sisteminin belleğine yasadışı olarak erişebileceğinden koruma sağlamaz; genel olarak, gerçek koruma için muhtemelen donanım desteği gereklidir [WL+93]. Diğer bir olumsuzluk ise bir kez yerleştirildikten sonra bir ad- dress alanının daha sonra başka bir yere taşınmasının zor olmasıdır [M65].

adres alanını fiziksel bellekte istediğimiz yere yerleştirebilir ve bunu yaparken sürecin yalnızca kendi adres alanına erişebilmesini sağlayabiliriz.

Bu kurulumda, her program sıfır adresine yüklenmiş gibi yazılır ve derlenir. Ancak, bir program çalışmaya başladığında, işletim sistemi fiziksel bellekte nereye yüklenmesi gerektiğine karar verir ve taban kaydını bu değere ayarlar. Yukarıdaki örnekte, işletim sistemi işlemi 32 KB fiziksel adrese yüklemeye karar verir ve böylece temel kaydı bu değere ayarlar.

Süreç çalışırken ilginç şeyler olmaya başlar. Şimdi, süreç tarafından herhangi bir bellek referansı oluşturulduğunda, bu işlemci tarafından aşağıdaki şekilde **çevrilir:**

```
fiziksel adres = sanal adres + taban
```

İşlem tarafından oluşturulan her bellek referansı **sanal** bir adrestir; donanım da bu adrese taban kaydının içeriğini ekler ve sonuç bellek sistemine verilebilecek **fiziksel bir adres olur.**

Bunu daha iyi anlamak için, tek bir komut yürütüldüğünde neler olduğunu izleyelim. Özellikle, önceki dizimizden bir komuta bakalım:

```
128: movl 0x0(%ebx), %eax
```

Program sayacı (PC) 128'e ayarlanmıştır; donanımın bu komutu alması gerektiğinde, 32896 fiziksel adresini elde etmek için önce değeri 32 KB (32768) temel kayıt değerine ekler; donanım daha sonra komutu bu fiziksel adresten alır. Ardından, işlemci talimatı yürütmeye başlar. Bir noktada, işlem daha sonra

İPUCU: DONANIM TABANLI DINAMIK YER DEĞIŞTIRME
Dinamik yeniden yerleştirme ile küçük bir donanım uzun bir yol kat eder. Yani, sanal adresleri (program tarafından oluşturulan) fiziksel adreslere dönüştürmek için bir **taban** yazmacı kullanılır. Bir **sınır** (ya da **limit**) kaydedicisi bu adreslerin adres alanının sınırları içinde kalmasını sağlar. Bunlar birlikte belleğin basit ve verimli bir şekilde sanallaştırılmasını sağlar.

sanal adres 15 KB'den yüklenir, işlemci bunu alır ve tekrar temel kayda (32 KB) ekler, 47 KB'lik nihai fiziksel adresi ve dolayısıyla istenen içeriği elde eder.

Sanal bir adresi fiziksel bir adrese dönüştürmek, tam olarak **adres çevirisi** olarak adlandırdığımız tekniktir; yani donanım, sürecin referans aldığını düşündüğü sanal bir adresi alır ve bunu verilerin gerçekte bulunduğu fiziksel b i r a d r e s e dönüştürür. Adresin bu şekilde yer değiştirmesi çalışma zamanında gerçekleştiğinden ve süreç çalışmaya başladıktan sonra bile adres alanlarını taşıyabildiğimizden, bu teknik genellikle **dinamik yer değiştirme olarak adlandırılır** [M65].

Şimdi şu soruyu soruyor olabilirsiniz: Sınırlar (limit) regülasyonuna ne oldu? Sonuçta, bu taban *ve sınırlar* yaklaşımı değil mi? Gerçekten de öyle. Tahmin edebileceğiniz gibi, bounds register korumaya yardımcı olmak için vardır. Özellikle, işlemci ilk olarak bellek referansının yasal olduğundan emin olmak için sınırlar *içinde olup* olmadığını kontrol edecektir; yukarıdaki basit örnekte, sınırlar kaydı her zaman 16 KB olarak ayarlanacaktır. Eğer bir süreç sınırlardan daha büyük ya da negatif bir gerçek adres üretirse, CPU bir istisna yaratacak ve süreç muhtemelen sonlandırılacaktır. Dolayısıyla sınırların amacı, süreç tarafından üretilen tüm adreslerin yasal ve sürecin "sınırları" içinde olduğundan emin olmaktır.

Taban ve sınır kayıtlarının çip üzerinde tutulan donanım yapıları olduğuna dikkat etmeliyiz (CPU başına bir çift). Bazen insanlar işlemcinin adres çevirisine yardımcı olan kısmına **bellek yönetim birimi (MMU)** adını verirler; daha sofistike bellek yönetim teknikleri geliştirdikçe MMU'ya daha fazla devre ekleyeceğiz.

İki yoldan biriyle tanımlanabilen bağlı kayıtlar hakkında küçük bir bilgi. Bir şekilde (yukarıdaki gibi), adres alanının boyutunu tutar ve böylece donanım tabanı eklemeden önce sanal adresi buna karşı kontrol eder. İkinci şekilde, adres alanının sonunun *fiziksel* adresini tutar ve böylece donanım önce tabanı ekler ve ardından adresin sınırlar içinde olduğundan emin olur. Her iki yöntem de mantıksal olarak eşdeğerdir; basitlik için genellikle ilk yöntemi varsayacağız.

## Örnek Çeviriler

Base-and-bounds aracılığıyla adres çevirisini daha ayrıntılı olarak anlamak için bir örneğe bakalım. Adres alanı 4 KB (evet, gerçekçi olmayan bir şekilde küçük) olan bir sürecin 16 KB fiziksel adrese yüklendiğini düşünün. İşte bir dizi adres çevirisinin sonuçları:

| Sanal Adres | | Fiziksel Adres |
|---|---|---|
| 0 | → | 16 KB |
| 1 KB | → | 17 KB |
| 3000 | → | 19384 |
| 4400 | → | *Fault (sınır dışı)* |

Örnekten de görebileceğiniz gibi, ortaya çıkan fiziksel adresi elde etmek için temel adresi sanal adrese (haklı olarak adres uzayında bir *ofset olarak görülebilir*) eklemeniz kolaydır. Yalnızca sanal adres "çok büyük" ya da negatifse sonuç bir hata olacak ve bir istisnanın ortaya çıkmasına neden olacaktır.

## 15.4 Donanım Desteği: Bir Özet

Şimdi donanımdan ihtiyacımız olan desteği özetleyelim (ayrıca bkz. Şekil 15.3, sayfa 9). İlk olarak, CPU sanallaştırma bölümünde tartışıldığı gibi, iki farklı CPU moduna ihtiyacımız vardır. İşletim sistemi, makinenin tamamına erişebildiği **ayrıcalıklı modda** (ya da **çekirdek modunda**) çalışır; uygulamalar ise yapabileceklerinin sınırlı olduğu **kullanıcı modunda** çalışır. Belki de bir tür **işlemci durum sözcüğünde** saklanan tek bir bit, CPU'nun o anda hangi modda çalıştığını gösterir; bazı özel durumlarda (örneğin, bir sistem çağrısı veya başka bir tür istisna veya kesme) CPU mod değiştirir.

Donanım ayrıca **taban ve sınır kayıtlarını** da sağlamalıdır. kendileri; bu nedenle her CPU, CPU'nun **bellek yönetim biriminin** (**MMU**) bir parçası olan ek bir çift kayda sahiptir. Bir kullanıcı programı çalıştığında, donanım her bir adresi, kullanıcı programı tarafından oluşturulan sanal adrese temel değeri ekleyerek çevirecektir. Donanım ayrıca adresin geçerli olup olmadığını kontrol edebilmelidir, bu da sınır kayıtları ve CPU içindeki bazı devreler kullanılarak gerçekleştirilir.

Donanım, taban ve sınır kayıtlarını değiştirmek için özel talimatlar sağlamalı ve işletim sisteminin farklı süreçler çalıştığında bunları değiştirmesine izin vermelidir. Bu talimatlar **ayrıcalıklıdır**; sadece çekirdek (ya da özel) modunda kayıtlar değiştirilebilir. Bir kullanıcı sürecinin[1] adresindeki taban kaydını keyfi olarak değiştirebilmesi durumunda yaratabileceği tahribatı hayal edin.

[1] "Yaratılabilecek" tahribat "tan başka bir şey var mı? [W17]

ASIDE: VERI YAPISI - THE FREE LIST

İşletim sistemi, süreçlere bellek tahsis edebilmek için boş belleğin hangi bölümlerinin kullanılmadığını takip etmelidir. Elbette böyle bir görev için birçok farklı veri yapısı kullanılabilir; en basit olanı (burada varsayacağımız), şu anda kullanılmayan fiziksel bellek aralıklarının bir **listesi olan bir boş listedir**.

| Donanım Gereksinimleri | Notlar |
|---|---|
| Ayrıcalıklı mod | *Kullanıcı modu süreçlerinin ayrıcalıklı işlemleri yürütmesini önlemek için gereklidir* |
| Taban/sınır kayıtları | *Adres çevirisini ve sınır kontrollerini desteklemek için CPU başına bir çift kayda ihtiyaç vardır* |
| Sanal adresleri çevirme yeteneği ve sınırlar içinde olup olmadığını kontrol edin | *Çevirileri yapmak ve kontrol etmek için devre sınırları; bu durumda, oldukça basit Ayrıcalıklı* talimat(*lar*) *İşletim sistemi bu değerleri ayarlayabilmelidir* güncelleme tabanı/sınırları *bir kullanıcı programının çalışmasına izin vermeden önce* |
| Kayıt için ayrıcalıklı talimat(lar) | *İşletim sistemi donanıma ne olduğunu söyleyebilmelidir* |
| istisna işleyicileri | *istisna oluşursa çalıştırılacak kod* |
| İstisnaları yükseltme yeteneği | *İşlemler ayrıcalıklı erişimlere erişmeye çalıştığında talimatlar veya sınır dışı bellek* |

Şekil 15.3: **Dinamik Yer Değiştirme: Donanım Gereksinimleri**

Koşuyorum. Hayal edin! Ve sonra bu tür karanlık düşünceleri zihninizden hızla uzaklaştırın, çünkü bunlar kabusların yapıldığı korkunç şeylerdir.

Son olarak, CPU, bir kullanıcı programının belleğe yasa dışı bir şekilde ("sınırların dışında" bir adresle) erişmeye çalıştığı durumlarda **istisnalar** üretebilmelidir; bu durumda CPU, kullanıcı programını yürütmeyi durdurmalı ve OS "sınırların dışında" **istisna işleyicisinin** çalışmasını ayarlamalıdır. İşletim sistemi işleyicisi daha sonra nasıl tepki vereceğini bulabilir, bu durumda muhtemelen süreci sonlandırır. Benzer şekilde, bir kullanıcı programı (ayrıcalıklı) taban ve sınır kayıtlarının değerlerini değiştirmeye çalışırsa, CPU bir istisna oluşturmalı ve "kullanıcı modundayken ayrıcalıklı bir işlem gerçekleştirmeye çalıştı" işleyicisini çalıştırmalıdır. CPU ayrıca bu işleyicilerin yerini bildirmek için bir yöntem sağlamalıdır; bu nedenle birkaç ayrıcalıklı talimat daha gereklidir.

## 15.5 İşletim Sistemi Sorunları

Donanımın dinamik yer değiştirmeyi desteklemek için yeni özellikler sunması gibi, işletim sisteminin de artık ele alması gereken yeni sorunlar vardır; donanım desteği ve işletim sistemi yönetiminin birleşimi basit bir sanal belleğin uygulanmasına yol açar. Özellikle, sanal belleğin temel ve sınırlı versiyonunu uygulamak için işletim sisteminin dahil olması gereken birkaç kritik nokta vardır.

İlk olarak, işletim sistemi bir süreç yaratıldığında harekete geçmeli ve bellekte adres alanı için yer bulmalıdır. Neyse ki, her adres alanının (a) fiziksel bellek boyutundan daha küçük olduğu ve (b) aynı boyutta, bu işletim sistemi için oldukça kolaydır; fiziksel belleği bir dizi yuva olarak görebilir ve her birinin boş mu yoksa kullanımda mı olduğunu takip edebilir. Yeni bir işlem oluşturulduğunda, işletim sisteminin yeni adres alanı için yer bulmak üzere bir veri yapısını (genellikle **boş liste** olarak adlandırılır) araması ve ardından kullanılmış olarak işaretlemesi gerekecektir. Değişken boyutlu adres

alanları ile hayat daha karmaşıktır, ancak bu konuyu gelecek bölümlere bırakacağız.

| İşletim Sistemi Gereksinimleri | Notlar |
|---|---|
| Bellek yönetimi | *Yeni işlemler için bellek ayırmanız gerekir; Sonlandırılan süreçlerden belleği geri alma; Genel olarak belleği **serbest liste** aracılığıyla yönetme* |
| Taban/sınır yönetimi | *Bağlam değiştirildiğinde taban/sınırları düzgün şekilde ayarlamalıdır* |
| İstisna işleme | *İstisnalar ortaya çıktığında çalıştırılacak kod; Muhtemel eylem, suç işleyen sürecin sonlandırılmasıdır* |

Şekil 15.4: **Dinamik Yer Değiştirme: İşletim Sistemi Sorumlulukları**

Bir örneğe bakalım. Şekil 15.2'de (sayfa 5), işletim sisteminin fiziksel belleğin ilk yuvasını kendisi için kullandığını ve yukarıdaki örnekteki işlemi fiziksel bellek adresi 32 KB'den başlayan yuvaya yerleştirdiğini görebilirsiniz. Diğer iki yuva boştur (16 KB-32 KB ve 48 KB-64 KB); dolayısıyla, **boş liste** bu iki girdiden oluşmalıdır.

İkinci olarak, işletim sistemi bir süreç sonlandırıldığında (yani, zarif bir şekilde çıktığında veya yanlış davrandığı için zorla öldürüldüğünde), diğer süreçlerde veya işletim sisteminde kullanılmak üzere tüm belleğini geri almak için bazı işler yapmalıdır. Bir süreç sonlandırıldığında, işletim sistemi bu sürecin belleğini serbest listeye geri koyar ve gerektiğinde ilişkili veri yapılarını temizler.

Üçüncü olarak, bir bağlam geçişi gerçekleştiğinde işletim sisteminin birkaç ek adım daha gerçekleştirmesi gerekir. Sonuçta, her CPU'da yalnızca bir taban ve sınır kayıt çifti vardır ve her program bellekte farklı bir fiziksel adrese yüklendiğinden, değerleri çalışan her program için farklıdır. Bu nedenle, işletim sistemi işlemler arasında geçiş yaptığında taban ve sınır çiftini *kaydetmeli ve geri yüklemelidir*. Özellikle, işletim sistemi bir süreci çalıştırmayı durdurmaya karar verdiğinde, taban ve sınır kayıtlarının değerlerini **süreç yapısı** veya süreç **kontrol bloğu** (PCB) gibi süreç başına bir yapıda belleğe kaydetmelidir. Benzer şekilde, işletim sistemi çalışan bir süreci devam ettirdiğinde (ya da ilk kez çalıştırdığında), CPU üzerindeki base ve bounds değerlerini bu süreç için doğru değerlere ayarlamalıdır.

Bir süreç durdurulduğunda (yani çalışmadığında), işletim sisteminin bir adres alanını bellekteki bir konumdan diğerine oldukça kolay bir şekilde taşımasının mümkün olduğunu belirtmeliyiz. Bir sürecin adres alanını taşımak için, işletim sistemi önce sürecin çizelgesini indirir; ardından, işletim sistemi adres alanını mevcut konumdan yeni konuma kopyalar; son olarak, işletim sistemi kaydedilen temel kaydı (süreç yapısında) yeni konuma işaret edecek şekilde günceller. Süreç yeniden başlatıldığında, (yeni) taban kaydı geri yüklenir ve talimatlarının ve verilerinin artık bellekte tamamen yeni bir noktada olduğundan habersiz olarak yeniden çalışmaya başlar.

Dördüncü olarak, işletim sistemi yukarıda tartışıldığı gibi **istisna işleyicileri** veya çağrılacak işlevler sağlamalıdır; işletim sistemi bu işleyicileri önyükleme zamanında yükler (ayrıcalıklı talimatlar yoluyla). Örneğin, bir süreç belleklere sınırları dışında erişmeye çalışırsa, CPU bir istisna yaratacaktır; işletim sistemi böyle bir istisna ortaya çıktığında harekete geçmeye hazır olmalıdır. İşletim sisteminin genel tepkisi düşmanca olacaktır: muhtemelen suç işleyen süreci sonlandıracaktır. İşletim sistemi çalıştırdığı makineye karşı son derece korumacı

olmalıdır ve bu nedenle belleğe erişmeye çalışan bir süreci hoş karşılamaz veya

| OS @ bootHardware Yok) (çekirdek modu) | (Henüz Program |
|---|---|
| **tuzak tablosunu başlat** | adreslerini hatırlayın...<br>sistem çağrısı işleyicisi<br>zamanlayıcı işleyicisi<br>yasadışı bellek erişimi işleyicisi<br>yasadışı komut işleyicisi |
| **kesme zamanlayıcısını başlat** | zamanlayıcıyı başlat; X ms sonra kesme |
| **süreç tablosunu başlatma serbest listeyi başlatma** | |

Şekil 15.5: **Sınırlı Doğrudan Yürütme (Dinamik Yer Değiştirme) @ Önyükleme**

yürütmemesi gereken talimatları yürütür. Güle güle yanlış davranan süreç; seni tanımak güzeldi.

Şekil 15.5 ve 15.6 (sayfa 12) donanım/OS etkileşiminin çoğunu bir zaman çizelgesinde göstermektedir. İlk şekil işletim sisteminin makineyi kullanıma hazırlamak için önyükleme sırasında ne yaptığını, ikincisi ise bir işlem (İşlem A) çalışmaya başladığında ne olduğunu göstermektedir; bellek aktarımlarının işletim sistemi müdahalesi olmadan donanım tarafından nasıl gerçekleştirildiğine dikkat edin. Bir noktada (ikinci şeklin ortasında), bir zamanlayıcı kesintisi meydana gelir ve işletim sistemi 'hatalı yükleme'(yasadışı bir bellek adresine) yapan Süreç B'ye geçer; bu noktada, işletim sistemi sürece dahil olmalı, süreci sonlandırmalı ve B'nin belleğini boşaltarak ve süreç tablosundan girişini kaldırarak temizlemelidir. Şekillerden de görebileceğiniz gibi, hala **sınırlı doğrudan yürütme** temel yaklaşımını takip ediyoruz. Çoğu durumda, işletim sistemi sadece donanımı uygun şekilde ayarlar ve sürecin doğrudan CPU üzerinde çalışmasına izin verir; sadece süreç yanlış davrandığında işletim sisteminin dahil olması gerekir.

## 15.6 Özet

Bu bölümde, sınırlı doğrudan yürütme kavramını, sanal bellekte kullanılan **ve adres çevirisi** olarak bilinen özel bir mekanizma ile genişlettik. Adres çevirisi ile işletim sistemi bir işlemden gelen her bir bellek erişimini kontrol edebilir ve erişimlerin adres alanı sınırları içinde kalmasını sağlayabilir. Bu tekniğin verimliliğinin anahtarı, her erişim için çeviriyi hızlı bir şekilde gerçekleştiren ve sanal adresleri (sürecin bellek görünümü) fiziksel adreslere (gerçek görünüm) dönüştüren donanım desteğidir. Tüm bunlar, yeri değiştirilen sürece aktarılacak şekilde gerçekleştirilir; sürecin bellek referanslarının çevrildiğinden haberi yoktur, bu da harika bir yanılsama yaratır. Ayrıca taban ve sınırlar ya da dinamik yer değiştirme olarak bilinen özel bir sanallaştırma biçimini de gördük. Taban ve sınırlar sanallaştırması oldukça *verimlidir*, çünkü bellekte bir yer değiştirme eklemek için yalnızca biraz daha fazla donanım mantığı gerekir.

| **İşletim sistemi @ run (çekirdek modu)** | **Donanım** | **Program (kullanıcı modu)** |
|---|---|---|
| **A işlemini başlatmak için:**<br>giriş tahsis et<br>süreç tablosunda süreç için bellek ayırma<br>temel/bağlı kayıtları ayarlama<br>**tuzaktan dönüş** (A'ya) | | |
| | A'nın kayıtlarını geri yükle<br>**kullanıcı moduna** geç<br>A'nın (başlangıç) bilgisayarına atla | **Süreç A çalışır**<br>Komut getirme<br>Komut sanal adresi çevir<br>getir<br>işlemini gerçekleştir<br>yürütme<br>açık yükleme/depolama<br>varsa: adresin yasal olduğundan emin olun<br>sanal adresi çevirin<br>yükleme/depolama gerçekleştirin<br>(Bir koşu...) |
| **Zamanlayıcı kolu**<br>karar: A'yı durdur, B'yi çalıştır<br>`switch()` rutinini çağır<br>regs(A)'yı kaydet<br>proc-struct(A)'ya (taban/sınırlar dahil)<br>regs(B)'yi geri yükle<br>from proc-struct(B)<br>(base/bounds dahil)<br>**return-from-trap** (into B) | **Zamanlayıcı kesmesi**<br>**çekirdek moduna** geç<br>işleyiciye atla | |
| | | **Süreç B çalışır**<br>Kötü yüklemeyi yürüt |
| **Tuzağı idare edin**<br>B sürecini öldürmeye karar vermek<br>B'nin belleğini boşaltmak<br>B'nin girişini boşaltmak<br>süreç tablosunda | B'nin kayıtlarını geri yükle<br>**kullanıcı moduna** geç<br>B'nin PC'sine atla<br>Yükleme sınırların dışında; **çekirdek moduna** geç<br>tuzak işleyicisine atla | |

Şekil 15.6: **Çalışma Zamanında Sınırlı Doğrudan Yürütme (Dinamik Yer Değiştirme)**

taban kaydını sanal adrese bağlar ve süreç tarafından oluşturulan adresin sınırlar içinde olup olmadığını kontrol eder. Base-and-bounds aynı zamanda *koruma* da sağlar; işletim sistemi ve donanım bir araya gelerek hiçbir sürecin kendi adres alanı dışında bellek referansları oluşturamamasını sağlar. Koruma kesinlikle işletim sisteminin en önemli hedeflerinden biridir; bu olmadan işletim sistemi makineyi kontrol edemez (eğer süreçler belleğin üzerine yazma özgürlüğüne sahip olsalardı, tuzak tablosunun üzerine yazmak ve sistemi ele geçirmek gibi kötü şeyleri kolayca yapabilirlerdi).

Ne yazık ki, bu basit dinamik yer değiştirme tekniğinin verimsizlikleri vardır. Örneğin, Şekil 15.2'de (sayfa 5) görebileceğiniz gibi, yeniden yerleştirilen süreç 32 KB'den 48 KB'ye kadar fiziksel bellek kullanmaktadır; ancak, süreç yığını ve heap çok büyük olmadığından, ikisi arasındaki tüm alan boşa *harcanmaktadır*. Bu tür israf genellikle **iç parçalanma** olarak adlandırılır, çünkü tahsis edilen birimin *içindeki* alanın tamamı kullanılmaz (yani parçalanır) ve dolayısıyla israf edilir. Mevcut yaklaşımımızda, daha fazla işlem için yeterli fiziksel bellek olmasına rağmen, şu anda bir adres alanını sabit boyutlu bir yuvaya yerleştirmekle kısıtlıyız ve bu nedenle iç parçalanma ortaya çıkabilir[2] . Bu nedenle, fiziksel belleği daha iyi kullanmaya çalışmak ve dahili parçalanmayı önlemek için daha sofistike makinelere ihtiyacımız olacak. İlk girişimimiz, daha sonra tartışacağımız **segmentasyon** olarak bilinen taban ve sınırların hafif bir genellemesi olacaktır.

[2]Bunun yerine farklı bir çözüm, adres alanı içine, kod bölgesinin hemen altına sabit boyutlu bir yığın ve bunun altına da büyüyen bir yığın yerleştirebilir. Ancak bu, özyineleme ve derin iç içe işlev çağrılarını zorlaştırarak esnekliği sınırlar ve bu nedenle kaçınmayı umduğumuz bir şeydir.

## References

[M65] "On Dynamic Program Relocation" by W.C. McGee. IBM Systems Journal, Volume 4:3, 1965, pages 184–199. *This paper is a nice summary of early work on dynamic relocation, as well as some basics on static relocation.*

[P90] "Relocating loader for MS-DOS .EXE executable files" by Kenneth D. A. Pillay. Microprocessors & Microsystems archive, Volume 14:7 (September 1990). *An example of a relocating loader for MS-DOS. Not the first one, but just a relatively modern example of how such a system works.*

[SS74] "The Protection of Information in Computer Systems" by J. Saltzer and M. Schroeder. CACM, July 1974. *From this paper: "The concepts of base-and-bound register and hardware-interpreted descriptors appeared, apparently independently, between 1957 and 1959 on three projects with diverse goals. At M.I.T., McCarthy suggested the base-and-bound idea as part of the memory protection system necessary to make time-sharing feasible. IBM independently developed the base-and-bound register as a mechanism to permit reliable multiprogramming of the Stretch (7030) computer system. At Burroughs, R. Barton suggested that hardware-interpreted descriptors would provide direct support for the naming scope rules of higher level languages in the B5000 computer system." We found this quote on Mark Smotherman's cool history pages [S04]; see them for more information.*

[S04] "System Call Support" by Mark Smotherman. May 2004. `people.cs.clemson.edu/~mark/syscall.html`. *A neat history of system call support. Smotherman has also collected some early history on items like interrupts and other fun aspects of computing history. See his web pages for more details.*

[WL+93] "Efficient Software-based Fault Isolation" by Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham. SOSP '93. *A terrific paper about how you can use compiler support to bound memory references from a program, without hardware support. The paper sparked renewed interest in software techniques for isolation of memory references.*

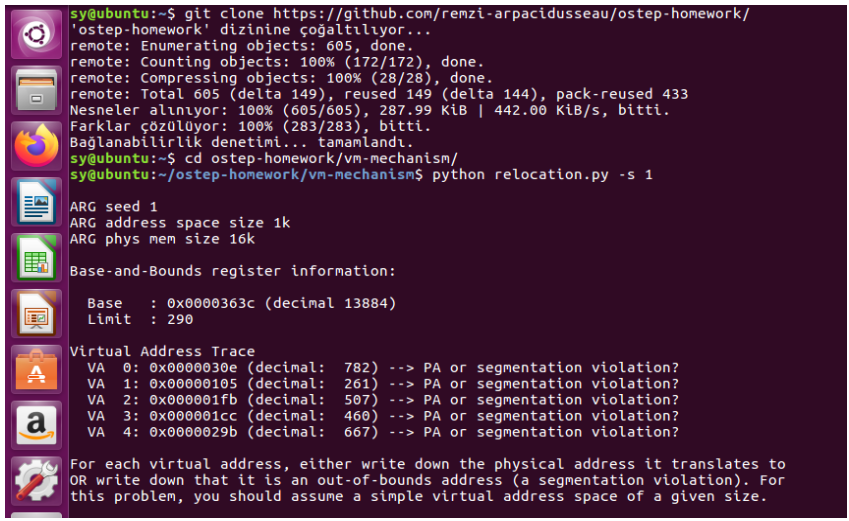
[W17] Answer to footnote: "Is there anything other than havoc that can be wreaked?" by Waciuma Wanjohi. October 2017. *Amazingly, this enterprising reader found the answer via google's Ngram viewing tool (available at the following URL: `http://books.google.com/ngrams`). The answer, thanks to Mr. Wanjohi: "It's only since about 1970 that 'wreak havoc' has been more popular than 'wreak vengeance'. In the 1800s, the word wreak was almost always followed by 'his/their vengeance'." Apparently, when you wreak, you are up to no good, but at least wreakers have some options now.*

## Ev Ödevi (Simülasyon)

`relocation.py` programı, taban ve sınır kayıtları olan bir sistemde adres geçişlerinin nasıl yapıldığını görmenizi sağlar. Ayrıntılar için README'ye bakın.

### Sorular

1. Tohum 1, 2 ve 3 ile çalıştırın ve işlem tarafından oluşturulan her sanal ad- dress'in sınırlar içinde mi yoksa dışında mı olduğunu hesaplayın. Sınırlar içindeyse, çeviriyi hesaplayın.



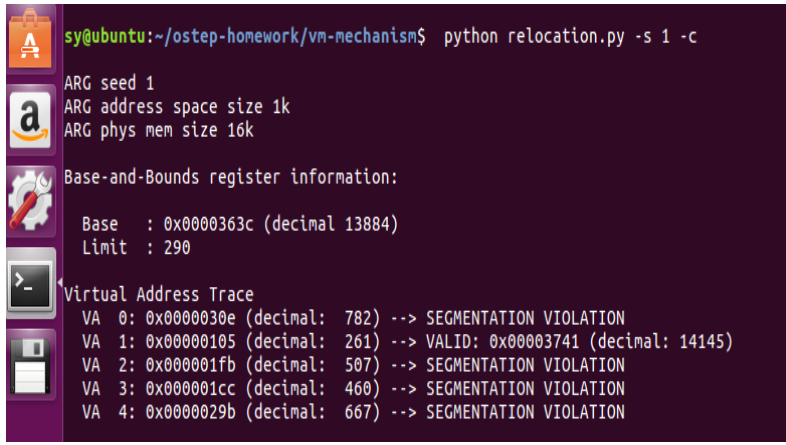

Yalnızca VA 1 sınırların dışında değildir.

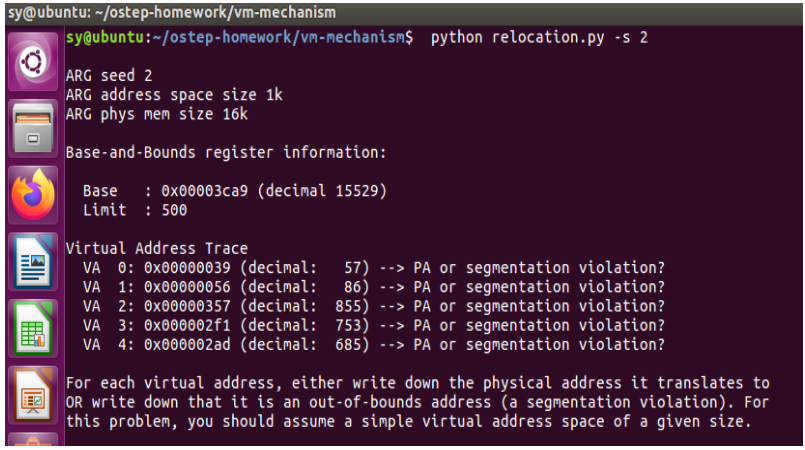
VA 1 --> GEÇERLİ: 0x00003741(ondalık: 14145)

VA N'nin ondalık basamağı (N = 0, 1, 2, 3, 4,...) < Limit olduğu sürece, limit aşılmaz.

Sınır dışı olmayanlar için:

VALID = Temel + Sanal Adres


sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 1 -c
ARG seed 1
ARG address space size 1k
ARG phys mem size 16k
Base-and-Bounds register information:
Base : 0x0000363c (decimal 13884)
Limit : 290
Virtual Address Trace
VA 0: 0x0000030e (decimal: 782) --> SEGMENTATION VIOLATION
VA 1: 0x00000105 (decimal: 261) --> VALID: 0x00003741 (decimal: 14145)
VA 2: 0x000001fb (decimal: 507) --> SEGMENTATION VIOLATION
VA 3: 0x000001cc (decimal: 460) --> SEGMENTATION VIOLATION
VA 4: 0x0000029b (decimal: 667) --> SEGMENTATION VIOLATION


sy@ubuntu: ~/ostep-homework/vm-mechanism
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 2
ARG seed 2
ARG address space size 1k
ARG phys mem size 16k
Base-and-Bounds register information:
Base : 0x00003ca9 (decimal 15529)
Limit : 500
Virtual Address Trace
VA 0: 0x00000039 (decimal: 57) --> PA or segmentation violation?
VA 1: 0x00000056 (decimal: 86) --> PA or segmentation violation?
VA 2: 0x00000357 (decimal: 855) --> PA or segmentation violation?
VA 3: 0x000002f1 (decimal: 753) --> PA or segmentation violation?
VA 4: 0x000002ad (decimal: 685) --> PA or segmentation violation?
For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.

```
sy@ubuntu:~/ostep-homework/vm-mechanism$  python relocation.py -s 2 -c

ARG seed 2
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x00003ca9 (decimal 15529)
  Limit  : 500

Virtual Address Trace
  VA  0: 0x00000039 (decimal:   57) --> VALID: 0x00003ce2 (decimal: 15586)
  VA  1: 0x00000056 (decimal:   86) --> VALID: 0x00003cff (decimal: 15615)
  VA  2: 0x00000357 (decimal:  855) --> SEGMENTATION VIOLATION
  VA  3: 0x000002f1 (decimal:  753) --> SEGMENTATION VIOLATION
  VA  4: 0x000002ad (decimal:  685) --> SEGMENTATION VIOLATION
```

```
sy@ubuntu:~/ostep-homework/vm-mechanism$  python relocation.py -s 3

ARG seed 3
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x000022d4 (decimal 8916)
  Limit  : 316

Virtual Address Trace
  VA  0: 0x0000017a (decimal:  378) --> PA or segmentation violation?
  VA  1: 0x0000026a (decimal:  618) --> PA or segmentation violation?
  VA  2: 0x00000280 (decimal:  640) --> PA or segmentation violation?
  VA  3: 0x00000043 (decimal:   67) --> PA or segmentation violation?
  VA  4: 0x0000000d (decimal:   13) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.
```

```
sy@ubuntu:~/ostep-homework/vm-mechanism$  python relocation.py -s 3 -c

ARG seed 3
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x000022d4 (decimal 8916)
  Limit  : 316

Virtual Address Trace
  VA  0: 0x0000017a (decimal:  378) --> SEGMENTATION VIOLATION
  VA  1: 0x0000026a (decimal:  618) --> SEGMENTATION VIOLATION
  VA  2: 0x00000280 (decimal:  640) --> SEGMENTATION VIOLATION
  VA  3: 0x00000043 (decimal:   67) --> VALID: 0x00002317 (decimal: 8983)
  VA  4: 0x0000000d (decimal:   13) --> VALID: 0x000022e1 (decimal: 8929)

sy@ubuntu:~/ostep-homework/vm-mechanism$
```

2. Bu bayraklarla çalıştırın: `-s 0 -n 10`. Hangi değeri ayarladınız oluşturulan tüm sanal adreslerin sınırlar dahilinde olduğundanemin olmak için `-l` (bounds register) değerini kullanabilirmiyim?

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 0 -n 10

ARG seed 0
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x00003082 (decimal 12418)
  Limit  : 472

Virtual Address Trace
  VA  0: 0x000001ae (decimal:  430) --> PA or segmentation violation?
  VA  1: 0x00000109 (decimal:  265) --> PA or segmentation violation?
  VA  2: 0x0000020b (decimal:  523) --> PA or segmentation violation?
  VA  3: 0x0000019e (decimal:  414) --> PA or segmentation violation?
  VA  4: 0x00000322 (decimal:  802) --> PA or segmentation violation?
  VA  5: 0x00000136 (decimal:  310) --> PA or segmentation violation?
  VA  6: 0x000001e8 (decimal:  488) --> PA or segmentation violation?
  VA  7: 0x00000255 (decimal:  597) --> PA or segmentation violation?
  VA  8: 0x000003a1 (decimal:  929) --> PA or segmentation violation?
  VA  9: 0x00000204 (decimal:  516) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.
```

Yukarıdaki resimden VA'daki en büyük değerin ondalık sayı 929 olduğunu görebiliriz, bu nedenle Limiti 930 olarak ayarlayabiliriz

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 0 -n 10 -l 929 -c

ARG seed 0
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x0000360b (decimal 13835)
  Limit  : 929

Virtual Address Trace
  VA  0: 0x00000308 (decimal:  776) --> VALID: 0x00003913 (decimal: 14611)
  VA  1: 0x000001ae (decimal:  430) --> VALID: 0x000037b9 (decimal: 14265)
  VA  2: 0x00000109 (decimal:  265) --> VALID: 0x00003714 (decimal: 14100)
  VA  3: 0x0000020b (decimal:  523) --> VALID: 0x00003816 (decimal: 14358)
  VA  4: 0x0000019e (decimal:  414) --> VALID: 0x000037a9 (decimal: 14249)
  VA  5: 0x00000322 (decimal:  802) --> VALID: 0x0000392d (decimal: 14637)
  VA  6: 0x00000136 (decimal:  310) --> VALID: 0x00003741 (decimal: 14145)
  VA  7: 0x000001e8 (decimal:  488) --> VALID: 0x000037f3 (decimal: 14323)
  VA  8: 0x00000255 (decimal:  597) --> VALID: 0x00003860 (decimal: 14432)
  VA  9: 0x000003a1 (decimal:  929) --> SEGMENTATION VIOLATION
```

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 0 -n 10 -l 930 -c

ARG seed 0
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x0000360b (decimal 13835)
  Limit  : 930

Virtual Address Trace
  VA  0: 0x00000308 (decimal:  776) --> VALID: 0x00003913 (decimal: 14611)
  VA  1: 0x000001ae (decimal:  430) --> VALID: 0x000037b9 (decimal: 14265)
  VA  2: 0x00000109 (decimal:  265) --> VALID: 0x00003714 (decimal: 14100)
  VA  3: 0x0000020b (decimal:  523) --> VALID: 0x00003816 (decimal: 14358)
  VA  4: 0x0000019e (decimal:  414) --> VALID: 0x000037a9 (decimal: 14249)
  VA  5: 0x00000322 (decimal:  802) --> VALID: 0x0000392d (decimal: 14637)
  VA  6: 0x00000136 (decimal:  310) --> VALID: 0x00003741 (decimal: 14145)
  VA  7: 0x000001e8 (decimal:  488) --> VALID: 0x000037f3 (decimal: 14323)
  VA  8: 0x00000255 (decimal:  597) --> VALID: 0x00003860 (decimal: 14432)
  VA  9: 0x000003a1 (decimal:  929) --> VALID: 0x000039ac (decimal: 14764)

sy@ubuntu:~/ostep-homework/vm-mechanism$
```

3. Bu bayraklarla çalıştırın: `-s 1 -n 10 -l 100`. Tabanın ayarlanabileceği maksimum değer nedir, öyle ki adres alanı fiziksel belleğin tamamına sığmaya devam etsin?

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 1 -n 10 -l 100

ARG seed 1
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x00000899 (decimal 2201)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00000363 (decimal:  867) --> PA or segmentation violation?
  VA  1: 0x0000030e (decimal:  782) --> PA or segmentation violation?
  VA  2: 0x00000105 (decimal:  261) --> PA or segmentation violation?
  VA  3: 0x000001fb (decimal:  507) --> PA or segmentation violation?
  VA  4: 0x000001cc (decimal:  460) --> PA or segmentation violation?
  VA  5: 0x0000029b (decimal:  667) --> PA or segmentation violation?
  VA  6: 0x00000327 (decimal:  807) --> PA or segmentation violation?
  VA  7: 0x00000060 (decimal:   96) --> PA or segmentation violation?
  VA  8: 0x0000001d (decimal:   29) --> PA or segmentation violation?
  VA  9: 0x00000357 (decimal:  855) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.

sy@ubuntu:~/ostep-homework/vm-mechanism$
```

Yukarıdaki şekle bir göz atalım, Limit değeri 100 ve fiziksel adresin değeri = Base + VA, fiziksel adresin temsil edilebilecek maksimum değeri = Base + Limit, yani Limit >= olduğu sürece VA, Base tarafından hangi değer ayarlanırsa ayarlansın, aralığı aşacaktır.
Bu nedenle, tüm adres alanlarının fiziksel belleğe sığması için ayarlanabilecek maksimum Base değeri yoktur.

4.Yukarıdaki aynı sorunlardan bazılarını çalıştırın, ancak daha büyük adres alanları (`-a`) ve fiziksel belleklerle (`-p`).

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 1 -n 10

ARG seed 1
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x0000363c (decimal 13884)
  Limit  : 290

Virtual Address Trace
  VA  0: 0x0000030e (decimal:  782) --> PA or segmentation violation?
  VA  1: 0x00000105 (decimal:  261) --> PA or segmentation violation?
  VA  2: 0x000001fb (decimal:  507) --> PA or segmentation violation?
  VA  3: 0x000001cc (decimal:  460) --> PA or segmentation violation?
  VA  4: 0x0000029b (decimal:  667) --> PA or segmentation violation?
  VA  5: 0x00000327 (decimal:  807) --> PA or segmentation violation?
  VA  6: 0x00000060 (decimal:   96) --> PA or segmentation violation?
  VA  7: 0x0000001d (decimal:   29) --> PA or segmentation violation?
  VA  8: 0x00000357 (decimal:  855) --> PA or segmentation violation?
  VA  9: 0x000001bb (decimal:  443) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.
```

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 1 -n 10 -a 2k

ARG seed 1
ARG address space size 2k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x0000363c (decimal 13884)
  Limit  : 580

Virtual Address Trace
  VA  0: 0x0000061c (decimal: 1564) --> PA or segmentation violation?
  VA  1: 0x0000020a (decimal:  522) --> PA or segmentation violation?
  VA  2: 0x000003f6 (decimal: 1014) --> PA or segmentation violation?
  VA  3: 0x00000398 (decimal:  920) --> PA or segmentation violation?
  VA  4: 0x00000536 (decimal: 1334) --> PA or segmentation violation?
  VA  5: 0x0000064f (decimal: 1615) --> PA or segmentation violation?
  VA  6: 0x000000c0 (decimal:  192) --> PA or segmentation violation?
  VA  7: 0x0000003a (decimal:   58) --> PA or segmentation violation?
  VA  8: 0x000006af (decimal: 1711) --> PA or segmentation violation?
  VA  9: 0x00000376 (decimal:  886) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.
```

Yukarıdaki iki resimden, adres alanı boyutu ikiye katlandığında (1K (varsayılan değer) --> 2K), karşılık gelen VA boyutunun da iki katına çıktığını görebiliriz.

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 1 -n 10 -p 32k

ARG seed 1
ARG address space size 1k
ARG phys mem size 32k

Base-and-Bounds register information:

  Base   : 0x00006c78 (decimal 27768)
  Limit  : 290

Virtual Address Trace
  VA  0: 0x0000030e (decimal:  782) --> PA or segmentation violation?
  VA  1: 0x00000105 (decimal:  261) --> PA or segmentation violation?
  VA  2: 0x000001fb (decimal:  507) --> PA or segmentation violation?
  VA  3: 0x000001cc (decimal:  460) --> PA or segmentation violation?
  VA  4: 0x0000029b (decimal:  667) --> PA or segmentation violation?
  VA  5: 0x00000327 (decimal:  807) --> PA or segmentation violation?
  VA  6: 0x00000060 (decimal:   96) --> PA or segmentation violation?
  VA  7: 0x0000001d (decimal:   29) --> PA or segmentation violation?
  VA  8: 0x00000357 (decimal:  855) --> PA or segmentation violation?
  VA  9: 0x000001bb (decimal:  443) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.
```

Fiziksel bellek boyutu iki katına çıkarıldığında (13884'ten (varsayılan değer) --> 27768), ilgili Tabanın boyutunun da iki katına çıktığını görebiliriz.

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 1 -n 10 -l 100

ARG seed 1
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x00000899 (decimal 2201)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00000363 (decimal:  867) --> PA or segmentation violation?
  VA  1: 0x0000030e (decimal:  782) --> PA or segmentation violation?
  VA  2: 0x00000105 (decimal:  261) --> PA or segmentation violation?
  VA  3: 0x000001fb (decimal:  507) --> PA or segmentation violation?
  VA  4: 0x000001cc (decimal:  460) --> PA or segmentation violation?
  VA  5: 0x0000029b (decimal:  667) --> PA or segmentation violation?
  VA  6: 0x00000327 (decimal:  807) --> PA or segmentation violation?
  VA  7: 0x00000060 (decimal:   96) --> PA or segmentation violation?
  VA  8: 0x0000001d (decimal:   29) --> PA or segmentation violation?
  VA  9: 0x00000357 (decimal:  855) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.

sy@ubuntu:~/ostep-homework/vm-mechanism$
```

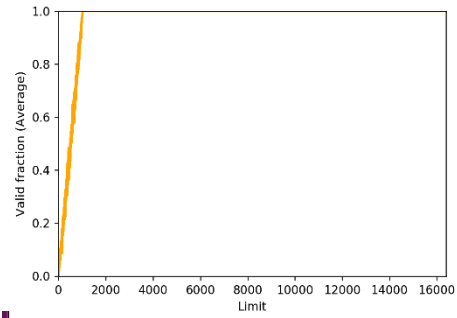
```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 1 -n 10 -l 100 -a 2k

ARG seed 1
ARG address space size 2k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x00000899 (decimal 2201)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x000006c7 (decimal: 1735) --> PA or segmentation violation?
  VA  1: 0x0000061c (decimal: 1564) --> PA or segmentation violation?
  VA  2: 0x0000020a (decimal:  522) --> PA or segmentation violation?
  VA  3: 0x000003f6 (decimal: 1014) --> PA or segmentation violation?
  VA  4: 0x00000398 (decimal:  920) --> PA or segmentation violation?
  VA  5: 0x00000536 (decimal: 1334) --> PA or segmentation violation?
  VA  6: 0x0000064f (decimal: 1615) --> PA or segmentation violation?
  VA  7: 0x000000c0 (decimal:  192) --> PA or segmentation violation?
  VA  8: 0x0000003a (decimal:   58) --> PA or segmentation violation?
  VA  9: 0x000006af (decimal: 1711) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.

sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 1 -n 100 -a 2k
```

Yukarıdaki iki resimden, adres alanı boyutu ikiye katlandığında (1K (varsayılan değer) --> 2K), karşılık gelen VA boyutunun da iki katına çıktığını görebiliriz.

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 1 -n 10 -l 100 -p 32k

ARG seed 1
ARG address space size 1k
ARG phys mem size 32k

Base-and-Bounds register information:

  Base   : 0x00001132 (decimal 4402)
  Limit  : 100

Virtual Address Trace
  VA  0: 0x00000363 (decimal:  867) --> PA or segmentation violation?
  VA  1: 0x0000030e (decimal:  782) --> PA or segmentation violation?
  VA  2: 0x00000105 (decimal:  261) --> PA or segmentation violation?
  VA  3: 0x000001fb (decimal:  507) --> PA or segmentation violation?
  VA  4: 0x000001cc (decimal:  460) --> PA or segmentation violation?
  VA  5: 0x0000029b (decimal:  667) --> PA or segmentation violation?
  VA  6: 0x00000327 (decimal:  807) --> PA or segmentation violation?
  VA  7: 0x00000060 (decimal:   96) --> PA or segmentation violation?
  VA  8: 0x0000001d (decimal:   29) --> PA or segmentation violation?
  VA  9: 0x00000357 (decimal:  855) --> PA or segmentation violation?

For each virtual address, either write down the physical address it translates to
OR write down that it is an out-of-bounds address (a segmentation violation). For
this problem, you should assume a simple virtual address space of a given size.
```

Fiziksel bellek boyutu iki katına çıkarıldığında (13884'ten (varsayılan değer) --> 27768), ilgili Tabanın boyutunun da iki katına çıktığını görebiliriz.

5.Sınırlar kaydının değerinin bir fonksiyonu olarak, rastgele üretilen sanal adreslerin ne kadarı geçerlidir? Farklı rastgele tohumlarla, 0'dan adres alanının maksimum boyutuna kadar değişen sınır değerleriyle çalışarak bir grafik oluşturun.




```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 0 -c

ARG seed 0
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x00003082 (decimal 12418)
  Limit  : 472

Virtual Address Trace
  VA  0: 0x000001ae (decimal:  430) --> VALID: 0x00003230 (decimal: 12848)
  VA  1: 0x00000109 (decimal:  265) --> VALID: 0x0000318b (decimal: 12683)
  VA  2: 0x0000020b (decimal:  523) --> SEGMENTATION VIOLATION
  VA  3: 0x0000019e (decimal:  414) --> VALID: 0x00003220 (decimal: 12832)
  VA  4: 0x00000322 (decimal:  802) --> SEGMENTATION VIOLATION
```

60%

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 1 -c

ARG seed 1
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x0000363c (decimal 13884)
  Limit  : 290

Virtual Address Trace
  VA  0: 0x0000030e (decimal:  782) --> SEGMENTATION VIOLATION
  VA  1: 0x00000105 (decimal:  261) --> VALID: 0x00003741 (decimal: 14145)
  VA  2: 0x000001fb (decimal:  507) --> SEGMENTATION VIOLATION
  VA  3: 0x000001cc (decimal:  460) --> SEGMENTATION VIOLATION
  VA  4: 0x0000029b (decimal:  667) --> SEGMENTATION VIOLATION
```

20%

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 2 -c

ARG seed 2
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x00003ca9 (decimal 15529)
  Limit  : 500

Virtual Address Trace
  VA  0: 0x00000039 (decimal:   57) --> VALID: 0x00003ce2 (decimal: 15586)
  VA  1: 0x00000056 (decimal:   86) --> VALID: 0x00003cff (decimal: 15615)
  VA  2: 0x00000357 (decimal:  855) --> SEGMENTATION VIOLATION
  VA  3: 0x000002f1 (decimal:  753) --> SEGMENTATION VIOLATION
  VA  4: 0x000002ad (decimal:  685) --> SEGMENTATION VIOLATION
```

40%

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 3 -c

ARG seed 3
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x000022d4 (decimal 8916)
  Limit  : 316

Virtual Address Trace
  VA  0: 0x0000017a (decimal:  378) --> SEGMENTATION VIOLATION
  VA  1: 0x0000026a (decimal:  618) --> SEGMENTATION VIOLATION
  VA  2: 0x00000280 (decimal:  640) --> SEGMENTATION VIOLATION
  VA  3: 0x00000043 (decimal:   67) --> VALID: 0x00002317 (decimal: 8983)
  VA  4: 0x0000000d (decimal:   13) --> VALID: 0x000022e1 (decimal: 8929)
```

40%

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s -5 -c

ARG seed -5
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x00002f79 (decimal 12153)
  Limit  : 415

Virtual Address Trace
  VA  0: 0x0000032e (decimal:  814) --> SEGMENTATION VIOLATION
  VA  1: 0x000003c5 (decimal:  965) --> SEGMENTATION VIOLATION
  VA  2: 0x000002f5 (decimal:  757) --> SEGMENTATION VIOLATION
  VA  3: 0x000003b0 (decimal:  944) --> SEGMENTATION VIOLATION
  VA  4: 0x0000001d (decimal:   29) --> VALID: 0x00002f96 (decimal: 12182)
```

20%

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 6 -c

ARG seed 6
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x0000349a (decimal 13466)
  Limit  : 459

Virtual Address Trace
  VA  0: 0x000001f0 (decimal:  496) --> SEGMENTATION VIOLATION
  VA  1: 0x0000010b (decimal:  267) --> VALID: 0x000035a5 (decimal: 13733)
  VA  2: 0x00000000 (decimal:    0) --> VALID: 0x0000349a (decimal: 13466)
  VA  3: 0x000002a6 (decimal:  678) --> SEGMENTATION VIOLATION
  VA  4: 0x000001e1 (decimal:  481) --> SEGMENTATION VIOLATION

sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 80 -c
```

60%

```
sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 6 -c

ARG seed 6
ARG address space size 1k
ARG phys mem size 16k

Base-and-Bounds register information:

  Base   : 0x0000349a (decimal 13466)
  Limit  : 459

Virtual Address Trace
  VA  0: 0x000001f0 (decimal:  496) --> SEGMENTATION VIOLATION
  VA  1: 0x0000010b (decimal:  267) --> VALID: 0x000035a5 (decimal: 13733)
  VA  2: 0x00000000 (decimal:    0) --> VALID: 0x0000349a (decimal: 13466)
  VA  3: 0x000002a6 (decimal:  678) --> SEGMENTATION VIOLATION
  VA  4: 0x000001e1 (decimal:  481) --> SEGMENTATION VIOLATION

sy@ubuntu:~/ostep-homework/vm-mechanism$ python relocation.py -s 80 -c
```

0%