15

Mekanizma: Adres Çevirimi

CPU sanallaştırmasını geliştirirken, limitli direkt çalıştırma (**limited direct execution (LDE))** olarak bilinen genel bir mekanizmaya odaklandık. LDE'nin arkasındaki fikir oldukça basittir: birçok kısım için, programın donanım üzerinde direkt olarak çalıştırılmasına izin verilir; buna rağmen, zaman içerisinde belirli önemli noktalarda (örneğin, bir işlem bir **sistem çağrısı (system call)** yaptığında veya bir **sayaç kesintisi (timer interrupt)** meydana geldiğinde), işletim sistemi dahil edilir ve "doğru" şeyin gerçekleştiğinden emin olunur. Böylece, işletim sistemi, biraz donanım desteği ile, verimli bir sanallaştırma sağlamak için çalışan programın yolundan çıkmak için elinden gelenin en iyisini yapar; ancak, zaman içindeki bu kritik noktalara müdahale ederek, işletim sistemi donanım üzerinde kontrolü elinde tutmasını sağlar. Verimlilik ve kontrol bir arada, herhangi bir modern işletim sisteminin ana hedeflerinden ikisidir.

Belleğin sanallaştırmada, istenen sanallaştırmayı sağlarken hem verimlilik hem de kontrol elde edinilerek benzer bir strateji izlenir. Verimlilik kazanmak için oldukça ilkel gibi gözüken donanım desteğinden yararlanmamız gerekir (örn., sadece birkaç **yazmaç (register)**) ama zamanla fazlasıyla karmaşıklaşacak (örn., TLBs, **sayfa-tablo (page-table)** desteği ve ileride karşılaşılacak benzerleri). Kontrol, işletim sisteminin hiçbir uygulamanın kendi belleği dışında herhangi bir belleğe erişmesine izin verilmemesini sağladığı anlamına gelir; bu nedenle uygulamaları kendilerinden ve işletim sistemini uygulamalardan korumak için burada da donanımdan yardıma ihtiyaç duyacağız. Son olarak, esneklik açısından **sanal makina (Virtual Machine)** sisteminden biraz daha fazlasına ihtiyacımız olacak; özellikle, programların adres alanlarını istedikleri şekilde kullanabilmelerini istiyoruz ve böylece sistemin programlanmasını kolaylaştırması sağlanacaktır. Böylece püf noktaya varıyoruz:

> PUF NOKTA: BELLEK NASIL VERİMLİ VE ESNEK BİÇİMDE SANALLAŞTIRILIR?
>
> Belleğin verimli bir sanallaştırmasını nasıl oluşturabiliriz? Uygulamaların ihtiyaç duyduğu esnekliği nasıl sağlıyoruz? Bir uygulamanın hangi bellek konumlarına erişebileceğinin kontrolünü nasıl sağlarız ve böylece uygulama belleği erişimlerinin uygun şekilde kısıtlanmasını nasıl sağlarız? Bütün bunları nasıl verimli bir şekilde yaparız?

Sınırlı doğrudan yürütme genel yaklaşımımıza bir ek olarak düşünebileceğiniz kullanacağımız genel teknik, **(donanım tabanlı adres çeviricisi (hardware-based address translation)** veya kısaca **adres çevirisi olarak (address translator)** olarak adlandırılan tekniktir. Adres çevirisi ile, donanım, talimat tarafından sağlanan **sanal (virtual)** adresi, istenen bilginin gerçekte bulunduğu **fiziksel (physical)** bir adresle değiştirerek, her bellek erişimini (örn., bir talimat getirme, yükleme veya depolama) fiziksel erişime dönüştürür. Bu nedenle, her bir bellek referansında, uygulama belleği referanslarını bellekteki gerçek konumlarına yeniden yönlendirmek için donanım tarafından bir adres çevirisi gerçekleştirilir.

Tabii ki, donanım tek başına belleği sanallaştıramaz, yalnızca bunu verimli bir şekilde yapmak için düşük seviyeli mekanizma sağlar. İşletim sistemi, doğru çevirilerin gerçekleşmesi için donanımı kurmak üzere önemli noktalara dahil olmalıdır.; bu nedenle **belleği yönetmeli (manage memory)**, hangi konumların boş, hangilerinin kullanımda olduğunu takip etmeli ve belleğin nasıl kullanıldığını kontrol etmek için akıllıca müdahale etmelidir.

Tekrardan söylemek gerekirse, tüm bu çalışmanın amacı güzel bir yanılsama yaratmaktır: programın, kendi kodunun ve verilerinin bulunduğu kendi özel belleğinin yaratılması. Bu sanal gerçekliğin arkasında çirkin fiziksel gerçek yatıyor: CPU (veya CPU'lar) bir programı çalıştırmak ve bir sonrakini çalıştırmak arasında geçiş yaptığından, birçok program aslında aynı anda belleği paylaşıyor. İşletim sistemi (donanımın yardımıyla) sanallaştırmayı kullanarak çirkin makine gerçekliğini kullanışlı, güçlü ve kullanımı kolay bir soyutlamaya dönüştürür.

15.1 Varsayımlar

Belleği sanallaştırmaya yönelik ilk girişimlerimiz çok basit olacak, hatta gülünç olacak. Durma, istediğin kadar gül; Çok yakında, TLB'lerin, çok seviyeli sayfa tablolarının ve diğer teknik harikaların içini ve dışını anlamaya çalıştığınızda, işletim sistemi size gülecek. İşletim sisteminin size gülmesi fikrinden hoşlanmıyor musunuz? Eh, o zaman şanssız olabilirsiniz; çünkü işletim sistemi böyle dönüyor.

Spesifik olarak, şimdilik kullanıcının adres alanının fiziksel belleğe *bitişik* olarak yerleştirilmesi gerektiğini varsayacağız. Ayrıca, basitlik için, adres alanının boyutunun çok büyük olmadığını varsayacağız; özellikle, *fiziksel belleğin boyutundan daha küçük* olmalıdır. Son olarak, her bir adres alanının tam olarak aynı boyutta olduğunu da varsayacağız. Bu varsayımlar gerçekçi gelmiyorsa endişelenmeyin; ilerledikçe onlardan bahsetmeyi bırakacağız ve hafızanın gerçekçi bir sanallaştırmasını gerçekleştireceğiz.

15.2 Örnek

Adres çevirisini uygulamak için ne yapmamız gerektiğini ve neden böyle bir mekanizmaya ihtiyacımız olduğunu daha iyi anlamak için basit bir örneğe bakalım. Adres alanı Şekil 15.1'de gösterildiği gibi olan bir süreç olduğunu hayal edin. Burada inceleyeceğimiz şey, hafızadan bir değer yükleyen, onu üç arttıran ve sonra değeri tekrar hafızaya

kaydeden kısa bir kod dizisidir. Bu kodun C dilindeki temsilinin şöyle görünebileceğini hayal edebilirsiniz:

İpucu: INTERPOZİSYON GUCLUDUR

İnterpozisyon, genellikle bilgisayar sistemlerinde büyük etki için kullanılan genel ve güçlü bir tekniktir. Belleği sanallaştırmada, donanım her bellek erişiminde araya girecek ve işlem tarafından verilen her sanal adresi, istenen bilgilerin fiilen depolandığı fiziksel bir adrese çevirecektir. Bununla birlikte, genel interpozisyon tekniği çok daha geniş bir şekilde uygulanabilir; gerçekten de yeni işlevler eklemek veya sistemin başka bir yönünü geliştirmek için hemen hemen her iyi tanımlanmış arabirim araya konulabilir. Böyle bir yaklaşımın olağan faydalarından biri şeffaflıktır; interpozisyon genellikle müşterinin arayüzü değiştirilmeden yapılır, bu nedenle söz konusu müşteride herhangi bir değişiklik gerektirmez.

```
void func() {
    int x = 3000; // thanks, Perry.
    x = x + 3;    // line of code we are interested in
    ...
```

Derleyici, bu kod satırını, bunun gibi görünebilecek bir derlemeye dönüştürür (x86 derlemesinde). `Disassemble`, Linux için `objdump` veya Mac için `otool` kullanılarak yapılır:

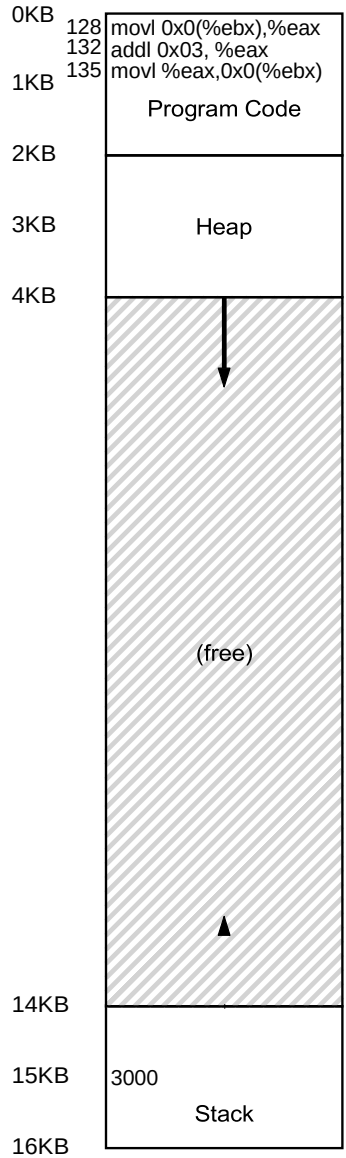
```
128: movl 0x0(%ebx), %eax     ;load 0+ebx into eax
132: addl $0x03, %eax         ;add 3 to eax register
135: movl %eax, 0x0(%ebx)     ;store eax back to mem
```

Bu kod parçacığı nispeten basittir; `x`'in adresinin `ebx` yazmaçına yerleştirildiğini varsayar ve sonra o adresteki değeri `movl` komutunu kullanarak genel amaçlı yazmaç `eax`'a yükler ("uzun kelime" hareketi için). Sonraki komut, `eax`'a 3 ekler ve son komut, `eax`'taki değeri aynı konumdaki belleğe geri kaydeder.

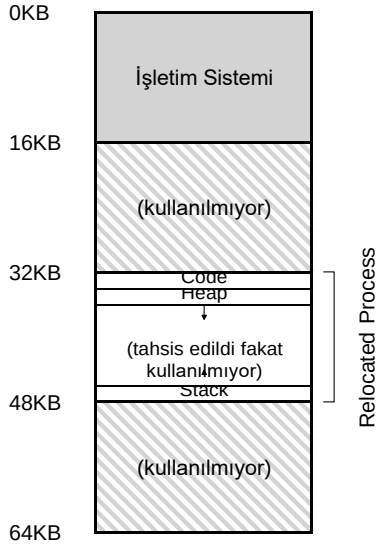
Şekil 15.1'de (sayfa 4), işlemin adres alanında hem kodun hem de verilerin nasıl düzenlendiğini gözlemleyin; üç komutlu kod dizisi 128 adresinde (kod bölümünde yukarıya yakın) ve `x` değişkeninin değeri 15 KB adresinde (alta yakın yığında) bulunur. Şekilde, yığındaki konumunda gösterildiği gibi, `x`'in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

Bu komutlar işlem açısından çalıştığında, aşağıdaki bellek erişimleri gerçekleşir.

- 128 adresindeki komutu getir
- Bu talimatı uygulayın (15 KB adresinden yükleyin)
- 132 adresindeki komutu getir
- Bu talimatı yürütün (bellek referansı yok)
- 135 adresindeki talimatı alın
- Bu talimatı yürütün (15 KB adresinde saklayın)



Figür 15.1: **Bir Süreç ve onun Adres Uzayı**



Figür 15.2: **Adreslenmiş Süreci İçeren Fiziksel Bellek**

Programın bakış açısından, adres alanı 0 adresinden başlar ve maksimum 16 KB'a kadar büyür; oluşturduğu tüm bellek referansları bu sınırlar içinde olmalıdır. Ancak, işletim sistemi belleği sanallaştırmak için işlemi mutlaka 0 adresinde değil, fiziksel bellekte başka bir yere yerleştirmek gerekir. Fakat bunu yaparken bir sorunumuz var: Bu süreci, **şeffaf** bir şekilde bellekte nasıl **yeniden** adresleyebiliriz? Gerçekte adres alanı başka bir fiziksel adresteyken, 0'dan başlayan sanal bir adres alanı yanılsamasını nasıl sağlayabiliriz? Bu işlem adres alanı belleğe yerleştirildikten sonra fiziksel belleğin nasıl görünebileceğine dair bir örnek Şekil 15.2'de bulunuyor. Şekilde, işletim sisteminin kendisi için ilk fiziksel bellek yuvasını kullandığını ve işlemi yukarıdaki örneğe göre 32 KB fiziksel bellek adresinden başlayarak yuvaya yerleştirdiğini görebilirsiniz. Diğer iki yuva boştur (16 KB-32 KB ve 48 KB-64 KB).

15.3 Dinamik (Donanım-tabanlı) Taşıma

Donanım tabanlı adres çevirisi hakkında biraz bilgi sahibi olmak için, ilk olarak ilk enkarnasyonunu tartışacağız. 1950'lerin sonlarındaki ilk zaman paylaşımlı makinelerde tanıtılan, **taban ve sınırlar (base and bounds)** olarak adlandırılan basit bir fikirdir; teknik aynı zamanda **dinamik taşıma** olarak da adlandırılır; ileride her iki terimi de birbirinin yerine kullanacağız [SS74].

Spesifik olarak, her biri içinde iki donanım yazmacına ihtiyacımız olacak: biri temel yazmaç, diğeri ise **sınırlar** (bazen **limit yazmacı (limit register)** olarak adlandırılır). Bu taban-sınır çifti,

ASIDE: YAZILIM-TABANLI YER DEĞİŞTİRME

İlk günlerde, donanım desteği ortaya çıkmadan önce, bazı sistemler yalnızca yazılım yöntemleriyle kaba bir yer değiştirme biçimini gerçekleştirdi. Temel teknik, **yükleyici (loader)** olarak bilinen bir yazılım parçasının çalıştırılmak üzere olan bir yürütülebilir dosyayı aldığı ve adreslerini fiziksel bellekte istenen ofsete yeniden yazdığı **statik yer değiştirme (static relocation)** olarak adlandırılır.

Örneğin, bir talimat 1000 adresinden bir yazmaca yüklenmişse (örn., `movl 1000, %eax`), ve programın adres alanı 3000 adresinden başlayarak yüklendiyse (programın düşündüğü gibi 0 değil), yükleyici talimatı her adresi 3000 ile dengelemek için yeniden yazacaktı (örn., `movl 4000, %eax`). Bu şekilde, işlemin adres alanının basit bir statik yer değiştirmesi sağlanır.

Ancak, statik yer değiştirmenin birçok sorunu vardır. İlk ve en önemlisi, işlemler kötü adresler üretebileceği ve dolayısıyla diğer işlemlerin ve hatta işletim sistemi belleğine ihlali olarak erişebileceği için koruma sağlamaz; genel olarak, gerçek koruma için büyük olasılıkla donanım desteği gerekir [WL+93]. Başka bir olumsuzluk, bir kez yerleştirildikten sonra bir adres alanını başka bir yere taşımanın zor olmasıdır. [M65].

adres alanını fiziksel bellekte istediğimiz herhangi bir yere yerleştirmemize izin verecek ve bunu işlemin yalnızca kendi adres alanına erişebilmesini sağlarken yapacağız.

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

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

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

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

Bunu daha iyi anlamak için, tek bir komut yürütüldüğünde ne olduğunu izleyelim. Spesifik olarak, önceki dizimizden bir talimata bakalım:

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

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

İpucu: DONANIM-TABANLI DİNAMIK ADRESLEME
Dinamik taşıma ile küçük bir donanım uzun bir yol kat eder. Yani, sanal adresleri (program tarafından oluşturulan) fiziksel adreslere dönüştürmek için bir **temel kayı (base register)** kullanılır. Bir **sınır (bound) kaydı**, bu adreslerin adres alanının sınırları içinde olmasını sağlar. Birlikte, belleğin basit ve verimli bir sanallaştırmasını sağlarlar.

15 KB sanal adresinden işlemcinin aldığı ve tekrar baz kayıt defterine (32 KB) eklediği yük, 47 KB'lik nihai fiziksel adresi ve dolayısıyla istenen içerikleri alır.

Sanal bir adresi fiziksel bir adrese dönüştürmek, tam olarak **adres çevirisi** olarak adlandırdığımız tekniktir; yani donanım, işlemin referans aldığını düşündüğü sanal bir adres alır ve onu, verilerin gerçekte bulunduğu fiziksel bir adrese dönüştürür. Adresin bu yer değiştirmesi çalışma zamanında gerçekleştiğinden ve süreç çalışmaya başladıktan sonra bile adres alanlarını taşıyabildiğimiz için, teknik genellikle **dinamik taşıma** [M65] olarak adlandırılır.

Şimdi şunu soruyor olabilirsiniz: Bu sınır (limit) kaydına ne oldu? Sonuçta, bu temel ve sınırlar yaklaşımı değil mi? Gerçekten öyle. Tahmin edebileceğiniz gibi, sınır kaydı korumaya yardımcı olmak için orada. Spesifik olarak, işlemci önce ihlali olmadığından emin olmak için bellek referansının sınırlar ve içinde olup olmadığını kontrol eder; yukarıdaki basit örnekte, sınır kaydı her zaman 16 KB'ye ayarlanacaktır. Bir işlem, sınırlardan daha büyük veya negatif bir sanal adres oluştursa, CPU'ya bir ihlal oluşturacak ve işlem büyük olasılıkla sonlandırılacaktır. Bu nedenle, sınırların amacı, süreç tarafından oluşturulan tüm adreslerin yasal ve sürecin "sınırları" içinde olduğundan emin olmaktır.

Taban ve sınır kayıtlarının çipte tutulan donanım yapıları olduğunu unutmamalıyız (her CPU başına bir çift). Bazen insanlar işlemcinin adres çevirisine yardımcı olan kısmını şu şekilde adlandırır: **bellek yönetim birimi** (**memory management unit) (MMU)**; daha karmaşık bellek yönetimi teknikleri geliştirildikçe, MMU'ya daha fazla devre ekleyeceğiz.

İki yoldan biriyle tanımlanabilen bağlı yazmaçlar hakkında küçük bir parça, birinci yol(yukarıdaki gibi), adres alanının boyutunu tutar ve bu nedenle donanım, tabanı eklemeden önce sanal adresi ona karşı kontrol eder. İkinci yol, 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 eski yöntemi kabul edeceğiz.

Örnek Dönüşümler

Taban ve sınırlar aracılığıyla adres çevirisini daha ayrıntılı anlamak için bir örneğe bakalım. 4 KB boyutunda (evet, gerçek olamayacak kadar küçük) bir adres alanına sahip bir işlemin 16 KB fiziksel adresine yüklendiğini hayal edin. İşte bu dizi adres çevirisinin sonuçları şu şekildedir:

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

Örnekte görebileceğiniz gibi, elde edilen fiziksel adresi elde etmek için temel adresi sanal adrese (doğru olarak adres alanına bir ofset olarak görüntülenebilir) eklemeniz kolaydır. Yalnızca sanal adres "çok büyük" veya negatifse, sonuç bir hata olur ve bir istisna oluşmasına neden olur.

15.4 Donanım Desteği: Özet

Şimdi donanımdan ihtiyacımız olan desteği özetleyelim (ayrıca bkz. Şekil 15.3, sayfa 9). İlk olarak, CPU sanallaştırma ile ilgili bölümde tartışıldığı gibi, iki farklı CPU moduna ihtiyacımız var. İşletim sistemi, tüm makineye erişiminin olduğu **ayrıcalıklı modda (privileged mode)** (veya **çekirdek modunda (kernel mode)**) çalışır; uygulamalar, yapabilecekleri sınırlı olan kullanıcı modunda çalışır. Belki de bir tür **işlemci durum sözcüğünde (processor status word)** saklanan tek bir bit, CPU'nun o anda hangi modda çalıştığını gösterir; belirli özel durumlarda (örneğin, bir sistem çağrısı veya başka tür bir istisna veya kesinti), CPU modları değiştirir.

Donanım ayrıca **taban ve sınır yazmaçları (base and bounds registers)** da sağlamalıdır; böylece her bir CPU, CPU'nun **bellek yönetim biriminin memory management unit** (**MMU**) bir parçası olan ek bir kayıt çiftine sahiptir. Bir kullanıcı programı çalışırken, donanım, kullanıcı programı tarafından oluşturulan sanal adrese temel değeri ekleyerek her adresi çevirecektir. Donanım aynı zamanda adresin geçerli olup olmadığını kontrol edebilmelidir, bu da sınır yazmacı ve CPU içindeki bazı devreler kullanılarak gerçekleştirilir.

Donanım, taban ve sınır kayıtlarını değiştirmek için özel talimatlar sağlamalı ve farklı işlemler çalıştığında işletim sisteminin bunları değiştirmesine izin vermelidir. Bu talimatlar **ayrıcalıklıdır (privileged)**; sadece çekirdek (veya ayrıcalıklı) modda kayıtlar değiştirilebilir. Bir kullanıcı işleminin, çalışırken temel yazmaci keyfi olarak değiştirebilmesi durumunda[1],

[1] "Yıkım" dan başka "yıkılabilecek" bir şey var mı? [W17]

KENAR NOTU: VERİ YAPISI — BOŞ LİSTE

İşletim sistemi, işlemlere bellek ayırabilmek için boş belleğin hangi bölümlerinin kullanılmadığını izlemelidir. Elbette böyle bir görev için birçok farklı veri yapısı kullanılabilir; en basiti (burada kabul edeceğiz), şu anda kullanımda olmayan fiziksel bellek aralıklarının bir listesi olan **boş bir (boş liste)** listedir.

Donanım İhtiyaçları	Notlar
Ayrıcalıklı Mod	*Kullanıcı modu işlemlerinin ayrıcalıklı işlemler yürütmesini önlemek için gerekli.*
Taban/sınır yazmaçları	*Adres çevirisini ve sınır kontrollerini desteklemek için CPU başına bir çift kayıt gerekir*
Sanal adresleri çevirebilme ve sınırlar içinde olup olmadığını kontrol edebilme	*Çeviri yapmak ve limitleri kontrol etmek için devreler; bu durumda oldukça basit*
Tabanı/sınırları güncellemek için ayrıcalıklı talimat(lar)	*OS, bir kullanıcı programının çalışmasına izin vermeden önce bu değerleri ayarlayabilmelidir.*
Kayıt için ayrıcalıklı talimat(lar) istisna işleyicileri	*İşletim sistemi, istisna oluşursa donanıma hangi kodun çalıştırılacağını söyleyebilmelidir*
İstisnaları yükseltme yeteneği	*İşlemler ayrıcalıklı erişmeye çalıştığında talimatlar veya sınır dışı bellek*

Figür 15.3: **Dinamik Taşıma: Donanım Gereksinimleri**

yaratabileceği tahribatı hayal edin. Hayal edin! ve sonra bu tür karanlık düşünceleri çabucak aklınızdan çıkarın, çünkü bunlar kabuslarda gerçekleşen korkunç şeylerdir.

Son olarak, bir kullanıcı programının belleğe ihlali olarak erişmeye çalıştığı durumlarda (bir adres "sınır dışı" olan) CPU'nun **istisnalar (exceptions)** oluşturabilmesi gerekir; bu durumda, CPU, kullanıcı programını yürütmeyi durdurmalı ve işletim sistemi "ihlali" **istisna işleyicisinin (exception handler)** çalışmasını sağlamalıdır. İşletim sistemi işleyicisi daha sonra nasıl tepki vereceğini anlayabilir, bu durumda muhtemelen süreci sonlandırabilir. Benzer şekilde, bir kullanıcı programı (ayrıcalıklı) taban ve sınır kayıtlarının değerlerini değiştirmeye çalışırsa, CPU bir istisna oluşturmalı ve "kullanıcı modundayken ayrıcalıklı bir işlem yürütmeye çalıştı" işleyicisini çalıştırmalıdır. CPU ayrıca, bu işleyicilerin konumu hakkında onu bilgilendirmek için bir yöntem sağlamalıdır; bu nedenle birkaç ayrıcalıklı talimat daha gereklidir.

15.5 İşletim Sistemi Sorunları

Donanımın dinamik taşıma desteklemek için yeni özellikler sağlaması gibi, işletim sisteminin de artık ele alması gereken yeni sorunları var; donanım desteği ve işletim sistemi yönetiminin birleşimi, basit bir sanal belleğin uygulanmasına yol açar. Spesifik olarak, sanal belleğin temel ve sınır sürümümüzü uygulamak için işletim sisteminin dahil olması gereken birkaç kritik nokta vardır. İlk olarak, işletim sistemi, bir işlem oluşturulduğunda, bellekte adres alanı için yer bularak harekete geçmelidir. Neyse ki, her bir 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 süreç oluşturulduğunda, işletim sisteminin yeni adres alanı için yer bulmak için bir veri yapısını (genellikle **boş liste (free list)**

olarak adlandırılır) araması ve ardından bunu kullanılmış olarak işaretlemesi gerekecektir. Değişken boyutlu adres alanları ile hayat daha karmaşıktır, ancak bu endişeyi gelecek bölümlere bırakacağız.

OS İhtiyaçları	Notlar
Bellek Yönetimi	*Yeni süreçler için alan tahsisine ihtiyaç duyar; Kaynağı, sonlandırılan süreçden geri alır; Genellikle belleği **boş listeler** ile yönetir.*
Taban/sınır yönetimi	*Bağlam anahtarında taban/sınırlar düzgün şekilde ayarlanmalıdır.*
İstisna yönetimi	*Çalışan kod istina uyandırdığında; muhtemel eylem süreci sonlandırmaktır.*

Figür 15.4: **Dinamik taşıma: İşletim Sistemi Sorumlukları**

Bir örneğe bakalım. Şekil 15.2'de (sayfa 5), işletim sisteminin kendisi için ilk fiziksel bellek yuvasını kullandığını ve işlemi yukarıdaki örnekteki gibi 32 KB fiziksel bellek adresinden başlayarak yuvaya yerleştirdiğini görebilirsiniz. Diğer iki yuva boştur (16 KB-32 KB ve 48 KB- 64 KB); bu nedenle, **boş liste (free list)** bu iki girişten oluşmalıdır.

İkincisi, işletim sistemi, bir işlem sonlandırıldığında (yani, zarif bir şekilde çıktığında veya yanlış davrandığı için zorla öldürüldüğünde), diğer işlemlerde veya işletim sisteminde kullanım için tüm belleğini geri alarak bazı işler yapmalıdır. Bir işlemin sona ermesi üzerine, işletim sistemi böylece belleğini boş listeye geri koyar ve gerektiğinde ilişkili veri yapılarını temizler.

Üçüncüsü, bir bağlam geçişi gerçekleştiğinde işletim sistemi ayrıca birkaç ek adım gerçekleştirmelidir. Sonuçta, her CPU'da yalnızca bir temel 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 sınırlar çiftini kaydetmeli ve geri yüklemelidir. Spesifik olarak, işletim sistemi bir işlemi çalıştırmayı durdurmaya karar verdiğinde, **işlem yapısı (process structer)** veya **process control block (işlem kontrol bloğu (PCB))** gibi işlem başına bazı yapılarda taban ve sınır kayıtlarının değerlerini belleğe kaydetmesi gerekir. Benzer şekilde, işletim sistemi çalışan bir işlemi sürdürdüğünde (veya ilk kez çalıştırdığında), CPU'daki taban ve sınırların değerlerini bu işlem için doğru değerlere ayarlamalıdır.

Bir işlem durdurulduğunda (yani çalışmadığında), işletim sisteminin bir adres alanını bellekteki bir konumdan diğerine oldukça kolay bir şekilde taşımasının mümkün olduğunu unutmamalıyız. Bir işlemin adres alanını taşımak için işletim sistemi önce işlemin zamanlamasını kaldırır; daha sonra işletim sistemi adres alanını geçerli konumdan yeni konuma kopyalar; son olarak, işletim sistemi kaydedilen temel kaydı (işlem yapısında) yeni konumu gösterecek şekilde günceller. İşlem yeniden başlatıldığında, (yeni) temel kaydı geri yüklenir ve komutlarının ve verilerinin artık bellekte tamamen yeni bir noktada olduğundan habersiz olarak yeniden çalışmaya başlar.

Dördüncüsü, işletim sistemi, yukarıda bahsedildiği gibi, istisna işleyicileri veya çağrılacak işlevleri sağlamalıdır; işletim sistemi bu işleyicileri önyükleme sırasında yükler (ayrıcalıklı talimatlar aracılığıyla). Örneğin, bir işlem kendi sınırları dışında belleğe erişmeye çalışırsa, CPU bir istisna oluşturacaktır; OS, böyle bir istisna ortaya çıktığında harekete geçmeye hazır olmalıdır. İşletim sisteminin ortak tepkisi düşmanlık olacaktır: büyük olasılıkla rahatsız edici süreci sonlandıracaktır. İşletim sistemi, çalıştırdığı makineye karşı son derece koruyucu olmalıdır ve bu nedenle, belleğe erişmeye veya

OS @ boot (kernel modu)	**Donanım**	**(Program Yok)**
kapan tablosunu başlat		
	Şu adressleri hatırla... Sistem çağrı yöneticisi zamanlayıcı yöneticisi ihlali bellek-erişimi yöneticisi	
kesinti zamanlayıcısı çalıştır	ihlali buyruk yöneticisi	
	zamanlayıcı çalıştır; X ms sonra kesinti	
süreç tablosunu oluştur **boş liste oluştur**		

Figür 15.5: **Limitli Direkt Çalıştırma (Dinamik Taşıma) @ Boot**

Yapmaması gereken talimatları yürütmeye çalışan bir sürece sıcak bakmaz. Güle güle, yaramazlık süreci; seni tanımak güzeldi.

Şekil 15.5 ve 15.6 (sayfa 12), bir zaman çizelgesindeki donanım/işletim sistemi etkileşiminin çoğunu göstermektedir. İlk şekil, işletim sisteminin önyükleme sırasında makineyi kullanıma hazırlamak için ne yaptığını gösterir ve ikincisi, bir süreç (İşlem A) çalışmaya başladığında ne olduğunu gösterir; bellek çevirilerinin donanım tarafından işletim sistemi müdahalesi olmadan nasıl işlendiğini not edin. Bir noktada (ikinci şeklin ortasında), bir zamanlayıcı kesintisi meydana gelir ve işletim sistemi, "kötü bir yük" (geçersiz bir bellek adresine) yürüten İşlem B'ye geçer; bu noktada, işletim sisteminin dahil olması, işlemi sonlandırması ve B'nin belleğini boşaltarak ve işlem tablosundan girişini kaldırarak temizlemesi gerekir. Rakamlardan da görebileceğiniz gibi, hala sınırlı doğrudan yürütme temel yaklaşımını izliyoruz. Çoğu durumda, işletim sistemi donanımı uygun şekilde kurar ve işlemin doğrudan CPU üzerinde çalışmasına izin verir; yalnızca süreç yanlış çalıştığında işletim sisteminin dahil olması gerekir.

Figür 15.6: **Limitli Direkt Çalıştırma (Dinamik Taşıma) @ Çalışma Zamanı**

OS @ run (kernel mode)	Hardware	Program (user mode)
A işlemini başlatmak için:		
Süreç tablosunda başlangıcı tahsis et		
süreç için bellek tahsis et		
taban/sınır kayıtlarını ayarla		
kapandan-geri-dön (A ya)		
	A'nın kayıtlarını geri yükle, kullanıcı moduna geç, A'nın (ilk) PC'sine atla	
		Süreç A çalışır
		Buyrukları elde et
	sanal adresi dönüştür güncellemeyi gerçekleştir	
		Burukları Çalıştır
	açık yükleme/depolama ise: adresin ihlalsiz olduğundan emin olun	
Sayacı Ele al	sanal adresi dönüştür	(A çalışır...)
decide: A'yı durdur, B'yi çalıştır	yükleme/depolamayı gerçekleştir	
`switch()` rutinini adresleri kayıt etmek için çağır(A) proc-struct(A) (base/bounds ile birlikte) proc-struct'dan yazmaçları geri yükle(B)	**Sayaç Kesintisi kernel moduna taşı** kontrolcüye atla	
(B) (base/bounds ile birlikte) **tuzaktan-geri-dön**(into B)		
	B'nin kayıtlarını geri getir. User moda geç. B'nin program sayacına atla.	**Süreç B çalışır**
		Kötü yükü çalıştır.
Tuzağı ele al		
B'yi sonlandırmaya karar ver	Yük sınır dışı;**kernel moda** geç ve tuzak ele alıcısını çalıştır.	
B'yi yeniden adresle		
Süreç tablosundan B girdisini temizle		

15.6 Özet

Bu bölümde, adres çevirisi olarak bilinen sanal bellekte kullanılan belirli bir mekanizma ile limitli doğrudan yürütme kavramını detayına indik. Adres çevirisi ile işletim sistemi, bir süreçten her bir bellek erişimini kontrol edebilir ve erişimlerin adres alanının sınırları içinde kalmasını sağlar. Bu teknik etkinliğinin anahtarı, sanal adresleri (sürecin bellek görünümü) fiziksel adreslere (gerçek görünüm) dönüştürerek her erişim için hızlı bir şekilde çeviri gerçekleştiren donanım desteğidir. Tüm bunlar, taşınan sürece şeffaf bir şekilde uygulanır; sürecin hafıza referanslarının tercüme edildiğinden haberi olmaz, bu da harika bir yanılsama yaratıyor. Ayrıca, taban ve sınırlar veya dinamik taşıma olarak bilinen belirli bir sanallaştırma biçimi gördük. Taban ve sınırlar sanallaştırması, sanal adrese bir temel kayıt eklemek ve süreç tarafından oluşturulan adresin sınırlar içinde olup olmadığını kontrol etmek için yalnızca biraz daha fazla donanım mantığı gerektiğinden oldukça verimlidir. Taban ve sınırlar ayrıca koruma sağlar; İşletim sistemi ve donanım, hiçbir işlemin kendi adres alanı dışında bellek referansları oluşturamaması için birleşir. Koruma, kesinlikle işletim sisteminin en önemli hedeflerinden biridir; onsuz, işletim sistemi makineyi kontrol edemezdi (işlemler belleğin üzerine yazmakta serbest olsaydı, tuzak tablosunun üzerine yazmak ve sistemi ele geçirmek gibi kötü şeyleri kolayca yapabilirlerdi).

Ne yazık ki, bu basit dinamik taşıma tekniğinin verimsizlikleri var. Örneğin, Şekil 15.2'de (sayfa 5) görebileceğiniz gibi, yeniden konumlandırılan işlem 32 KB'den 48 KB'ye kadar fiziksel bellek kullanıyor; bununla birlikte, süreç yığını ve yığını çok büyük olmadığından, ikisi arasındaki boşluğun tamamı boşa harcanıyor. Bu tür atıklara genellikle dahili parçalanma denir, çünkü tahsis edilen birim içindeki alanın tamamı kullanılmaz (yani parçalanır) ve bu nedenle boşa harcanır. Mevcut yaklaşımımızda, daha fazla işlem için yeterli fiziksel bellek olmasına rağmen, şu anda sabit boyutlu bir yuvaya bir adres alanı yerleştirmekle sınırlıyız ve bu nedenle iç parçalanma ortaya çıkabilir[2]. Bundan dolayı, fiziksel belleği daha iyi kullanmak ve dahili parçalanmadan kaçınmak için daha gelişmiş makinelere ihtiyacımız olacak. İlk girişimimiz, daha sonra tartışacağımız, segmentasyon olarak bilinen taban ve sınırların hafif bir genellemesi olacaktır.

[2]Bunun yerine farklı bir çözüm olarak adres alanına, kod bölgesinin hemen altına sabit boyutlu bir yığın ve bunun altında büyüyen bir yığın yerleştirebilir. Ancak bu, özyinelemeyi ve derinlemesine iç içe işlev çağrılarını zorlayarak esnekliği sınırlar ve bu nedenle kaçınmayı umduğumuz bir durumdur.

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.*

Ödev (Simülasyon)

`relocation.py` programı adres dönüşümlerinin sistem içerisinde taban/sınır ile birlikte nasıl çalıştığını görmenizi sağlar. Detaylar için README dosyasını inceleyiniz.

Sorular

1. Programı 1, 2 ve 3 seedleri ile çalıştırın ve süreç tarafından oluşturulan her bir sanal adresin sınırların içinde mi yoksa dışında mı olduğunu hesaplayın. Sınırlar içindeyse, çeviriyi hesaplayın.

Çıktı:

```
➜ vm-mechanism (master) ✔ python relocation.py -s 1

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?
```

```
➜ vm-mechanism (master) ✔ 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?
```

```
➜ vm-mechanism (master) ✓ 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?
```

Çözüm:

İstenilen cevap için değerlendireceğimiz değer limit değeridir. Sanal adres (VA X) değeri eğer bounds register dışına çıkmıyorsa sanal adres VA X geçerlidir aksi halde segmentasyon ihlaline girerek sürecin kendi için tahsis edilen alan dışına çıkmaktadır.

Seed : 1

VA 1 dışındaki tüm sanal bellekler ihlale sebep olmaktadır.

VA 1 : 0x0000363c | 0x00000105 = 0x00003741 adresine sahiptir.

Seed : 2

VA 0 ve VA 1 dışındaki tüm sanal bellekler ihlale sebep olmaktadır.

VA 0 : 0x00003ca9 | 0x00000039 = 0x00003ce2 adresine sahiptir.

VA 1 : 0x00003ca9 | 0x00000056 = 0x00003cff adresine sahiptir.

Seed : 3

VA 3 ve VA 4 dışındaki tüm sanal bellekler ihlale sebep olmaktadır.

VA 3 : 0x000022d4 | 0x00000043 = 0x00002317 adresine sahiptir.

VA 4 : 0x000022d4 | 0x0000000d = 0x000022e1 adresine sahiptir.

2. Şu parametrelerle çalıştırın: `-s 0 -n 10`. Oluşturulacak tüm sanal adreslerin sınırlar içerisinde kalmasını garantilemek için `-l` (sınır kaydı) parametresiyle sınır kaydını nasıl ayarladınız?

Çıktı :

```
➜ vm-mechanism (master) ✓ 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.
```

Çözüm:

Betiği -s 0 ve -n 10 parametreleriyle çalıştırdığımda yukarıdaki çıktı üretilmektedir. Bu çıktı incelendiğinde en büyük sınırlara sahip sanal adresin VA 8 (0x000003a1) olduğu görülmektedir. Bu adresin decimal değeri 929'dur. Dolayısıyla -l parametresi ile ayarlayacağımı sınır kaydı 929'dan büyük bir değer olması şartıyla ihlallerin önüne geçebiliriz.

3. Şu parametrelerle çalıştırın: `-s 1 -n 10 -l 100`. Adres alanı hala bütünüyle fiziksel belleğe sığması şartıyla, tabanın ayarlanabileceği maksimum değer nedir?

Çıktı:

```
➜ vm-mechanism (master) ✓ 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.
```

Çözüm:
Burada kontrol edeceğimiz nokta taban kaydımızın en büyük sanal adresten (VA X) küçük olmamasıdır. Çıktıyı incelediğimizde VA 0 (0x00000363) en büyük değere sahip adrestir. Dolayısıyla tabanın alabileceği en büyük değer 0x00000364 (decimal : 868) olacaktır.

4. Yukarıdaki bazı soruları tekrar çalıştırın, fakat daha büyük bir adres alanıyla(`-a`) ve fiziksel bellekle(`-p`).

```
➜ vm-mechanism (master) ✓ python relocation.py -a 32m -p 64m -s 0

ARG seed 0
ARG address space size 32m
ARG phys mem size 64m

Base-and-Bounds register information:

  Base   : 0x03082532 (decimal 50865458)
  Limit  : 15472131

Virtual Address Trace
  VA  0: 0x00d75528 (decimal: 14112040) --> PA or segmentation violation?
  VA  1: 0x008490bc (decimal: 8687804) --> PA or segmentation violation?
  VA  2: 0x0105c5cc (decimal: 17155532) --> PA or segmentation violation?
  VA  3: 0x00cf5386 (decimal: 13587334) --> PA or segmentation violation?
  VA  4: 0x01914e0c (decimal: 26299916) --> PA or segmentation violation?
```

```
vm-mechanism (master) ✓ python relocation.py -a 32m -p 64m -s 0 -n 10

ARG seed 0
ARG address space size 32m
ARG phys mem size 64m

Base-and-Bounds register information:

  Base   : 0x03082532 (decimal 50865458)
  Limit  : 15472131

Virtual Address Trace
  VA  0: 0x00d75528 (decimal: 14112040) --> PA or segmentation violation?
  VA  1: 0x008490bc (decimal: 8687804) --> PA or segmentation violation?
  VA  2: 0x0105c5cc (decimal: 17155532) --> PA or segmentation violation?
  VA  3: 0x00cf5386 (decimal: 13587334) --> PA or segmentation violation?
  VA  4: 0x01914e0c (decimal: 26299916) --> PA or segmentation violation?
  VA  5: 0x009b4bce (decimal: 10177486) --> PA or segmentation violation?
  VA  6: 0x00f40484 (decimal: 15991940) --> PA or segmentation violation?
  VA  7: 0x012ab10c (decimal: 19575052) --> PA or segmentation violation?
  VA  8: 0x01d0f42c (decimal: 30471212) --> PA or segmentation violation?
  VA  9: 0x01026650 (decimal: 16934480) --> PA or segmentation violation?
```

Çözüm :
Burada iki örnek yaptım, ilk örnek ilk soruya ait seed 0 ile yapılması istenen örnekti. İkincisi ise ikinci soruya ait örnektir.

1. Örnek

Başlangıçta :

VA 1 dışındaki tüm sanal bellekler ihlale sebep olmaktadır.
VA 1 : 0x0000363c | 0x00000105 = 0x00003741 adresine sahiptir.

Adres alanı ve fiziksel alan büyütülünce:

VA 0, VA 1, VA 3 dışındaki tüm sanal bellekler ihlale sebep olmaktadır.
VA 1 : 0x03082532 | 0x00d75528 = 0x03df7a5a adresine sahiptir.
VA 2 : 0x03082532 | 0x008490bc = 0x038cb5ee adresine sahiptir.
VA 3 : 0x03082532 | 0x00cf5386 = 0x03d778b8 adresine sahiptir.

2. Örnek

Başlangıçta:
Betiği -s 0 ve -n 10 parametreleriyle çalıştırdığımda yukarıdaki çıktı üretilmektedir. Bu çıktı incelendiğinde en büyük sınırlara sahip sanal adresin VA 8 (0x000003a1) olduğu görülmektedir. Bu adresin decimal değeri 929'dur. Dolayısıyla -l parametresi ile ayarlayacağımı sınır kaydı 929'dan büyük bir değer olması şartıyla ihlallerin önüne geçebiliriz.

Adres alanı ve fiziksel alan büyütülünce:
Betiği -s 0 ve -n 10 parametreleriyle çalıştırdığımda yukarıdaki çıktı üretilmektedir. Bu çıktı incelendiğinde en büyük sınırlara sahip sanal adresin VA 8 (0x01d0f42c) olduğu görülmektedir. Bu adresin decimal değeri 30471212'dir. Dolayısıyla -l parametresi ile ayarlayacağımı sınır kaydı 30471212'dan büyük bir değer olması şartıyla ihlallerin önüne geçebiliriz.

5. Sınır kaydının değerinin bir fonksiyonu olarak, rastgele oluşturulmuş sanal adreslerin hangi kısmı geçerlidir? 0'dan adres alanının maksimum boyutuna kadar değişen sınır değerlerle farklı rastgele tohumlarla çalışan bir grafik yapın.

Çözüm:
Ödevin bu kısmını yapabilmek için relocation.py dosyasında bazı değişiklikler yapmam gerekti. Bu değişikler 1,000,000 adet sanal adres üretmesi ve taşıp taşmadığını kontrol etmesiydi. Daha sonra kontrol ettikleri arasından bir oran çıkarıp bu oranı konsola yazdırırken zamana bağlı ihlalleride grafik haline getirip görselleştiriyor.

1,000,000 sanal adreste ihlal oranı : 0.5375'dir.