16

Segmentasyon

Şimdiye kadar her işlemin tüm adres alanını belleğe koyuyorduk. Taban ve sınır kayıtlarıyla, İşletim Sistemi işlemleri fiziksel belleğin farklı bölümlerine kolayca taşıyabilir. Bununla birlikte, bu adres alanlarımız hakkında ilginç bir şey fark etmiş olabilirsiniz: tam ortada, Stack ve Heap bellek arasında büyük bir "boş" alan yığını var.

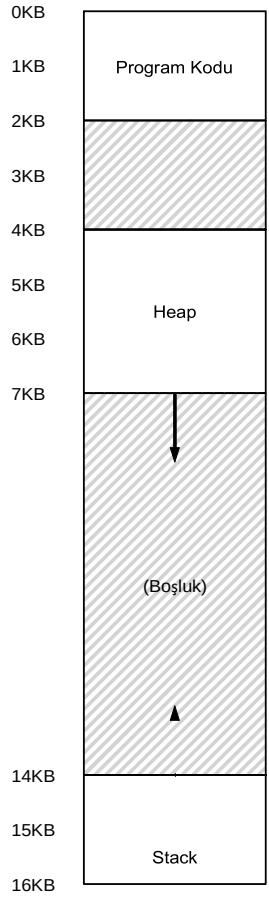
Şekil 16.1'den de tahmin edebileceğiniz gibi, Stack ve Heap arasındaki boşluk **işlem (process)** tarafından kullanılmamasına rağmen, tüm adres alanını fiziksel bellekte bir yere yeniden yerleştirdiğimizde yine de fiziksel belleği kaplıyor ;bu nedenle, belleği sanallaştırmak için taban ve sınırlar kayıt çifti kullanmanın basit yaklaşımı israftır. Ayrıca, tüm adres alanı belleğe sığmadığında bir programı çalıştırmayı oldukça zorlaştırır ; Bu nedenle, taban ve sınırlar istediğimiz kadar esnek değildir. Ve böylece:

Püf Noktası: Büyük Bir Adres Alanı Nasıl Desteklenir

Stack ve Heap arasında (potansiyel olarak) çok fazla boş alan bulunan büyük bir adres alanını nasıl destekleriz? Örneklerimizde, küçük (taklit edilmiş) adres alanlarında, atıkların çok kötü görünmediğini unutmayın. Bununla birlikte, 32 bitlik bir adres alanı (4 GB boyutunda) düşünün; tipik bir program yalnızca megabaytlarca bellek kullanır, ancak yine de tüm adres alanının bellekte yerleşik olmasını ister.

16.1 Segmentasyon: Genelleştirilmiş Taban/Sınırlar

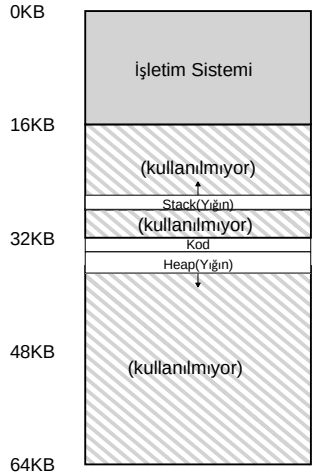
Bu sorunu çözmek için bir fikir doğdu ve buna **segmentation (segmentasyon)** adı verildi. Bu oldukça eski bir fikirdir, en azından 1960'ların başlarına kadar uzanır [H61, G62]. Fikir basit: MMU'muzda yalnızca bir taban ve sınır çiftine sahip olmak yerine, neden adres alanının mantıksal **segment(bölüm)** başına bir taban ve sınır çifti olmasın? Bir segment, belirli bir uzunluktaki adres alanının yalnızca bitişik bir kısmıdır,



Şekil 16.1: **Bir Adres Alanı (Tekrar)**

Ve kurallı adres alanımızda, mantıksal olarak farklı üç segmentimiz var: kod, Stack ve Heap . Segmentasyonun işletim sisteminin yapmasına izin verdiği şey, bu segmentlerin her birini fiziksel belleğin farklı bölümlerine yerleştirmek ve böylece fiziksel belleği kullanılmayan sanal adres alanıyla doldurmaktan kaçınmaktır.

Bir örneğe bakalım. Adres alanını Şekil 16.1'deki fiziksel belleğe yerleştirmek istediğimizi varsayalım. Segment başına bir taban ve sınır çifti ile , her segmenti *bağımsız olarak* fiziksel belleğe yerleştirebiliriz. Örneğin, bkz. Şekil 16.2'ye (sayfa 3);Burada içinde bu üç segmentin bulunduğu 64 KB'lık bir fiziksel bellek görürsünüz (ve işletim sistemi için ayrılmış 16 KB).



Şekil 16.2: **Segmentleri Fiziksel Belleğe Yerleştirme**

Diyagramda görebileceğiniz gibi, fiziksel bellekte yalnızca kullanılan belleğe yer ayrılır ve bu nedenle büyük miktarda kullanılmayan adres alanına sahip(bazen **sparse address spaces** (seyrek adres alanları) olarak adlandırdığımız) büyük adres alanları yerleştirilebilir.

MMU'muzdaki segmentasyonu desteklemek için gereken donanım yapısı tam da beklediğiniz şeydir: bu durumda, üç taban ve sınırdan oluşan bir dizi çifti kaydeder. Aşağıdaki Şekil 16.3, yukarıdaki sınav için kayıt değerlerini göstermektedir ; Her sınır kaydı bir segmentin boyutunu tutar .

Segment	Taban	Boyutu
Kod	32K	2K
Heap	34K	3K
Stack	28K	2K

Şekil 16.3: **Segment Kayıt Değerleri**

Şekilden, kod segmentinin 32KB fiziksel adrese yerleştirildiğini, 2KB boyutuna sahip olduğunu ve Heap segmentinin 34KB'ye yerleştirildiğini ve 3KB boyutuna sahip olduğunu görebilirsiniz. Buradaki boyut segmenti, daha önce tanıtılan sınır kaydıyla tamamen aynıdır; donanıma bu segmentte tam olarak kaç baytın geçerli olduğunu söyler (ve böylece donanımın bir programın bu sınırların dışında ne zaman yasa dışı erişim yaptığını belirlemesini sağlar).

Şekil 16.1'deki adres alanını kullanarak örnek bir çeviri yapalım . Sanal adres 100'e bir referans yapıldığını varsayalım (Şekil 16.1, sayfa 2'de görsel olarak görebileceğiniz gibi kod segmentinde olan).

Bir yana: Segmentasyon Hatası

Segmentation fault (segmentasyon hatası) veya ihlali terimi, segmentlere ayrılmış bir makinede geçersiz bir adrese yapılan bellek erişiminden kaynaklanır. Komik bir şekilde, segmentasyonu hiç desteklemeyen makinelerde bile terim devam ediyor. Veya kodunuzun neden sürekli hatalı olduğunu anlamıyorsanız , o kadar komik değil.

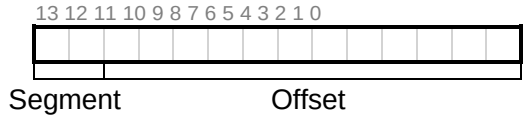
Referans gerçekleşir (örneğin, bir komut getirme işleminde), donanım istenen fiziksel adrese ulaşmak için *bu segmentteki*(bu durumda 100) ofsete temel değeri ekleyecektir: 100 + 32KB veya 32868. Daha sonra adresin sınırlar içinde olup olmadığını kontrol eder (100, 2 KB'den küçüktür), öyle olduğunu bulur ve 32868 fiziksel bellek adresine referans verir.

Şimdi Heap'deki bir adrese bakalım, sanal adres 4200 (yine Şekil 16.1'e bakınız). Heap tabanına (34 KB) 4200 sanal adresini eklersek, doğru fiziksel adres olmayan 39016 fiziksel adresini elde ederiz. Öncelikle yapmamız gereken, ofseti yığına çıkarmak, yani adresin bu segmentteki hangi bayt(lar)a atıfta bulunduğudur. Heap sanal adres 4 KB (4096) 'de başladığından, 4200 ofseti aslında 4200 eksi 4096 veya 104'tür. Daha sonra bu ofseti (104) alır ve istenen sonucu elde etmek için temel kayıt fiziksel adresine (34K) ekleriz: 34920.

Yığının(Heap) sonunun ötesinde olan geçersiz bir adrese (yani, 7KB veya daha büyük bir sanal adres) atıfta bulunmaya çalışırsak ne olur? Ne olacağını hayal edebilirsiniz: donanım, adresin sınırların dışında olduğunu algılar, işletim sistemine hapsolur ve muhtemelen rahatsız edici sürecin sonra ermesine yol açar ve artık tüm C programcıların korkmayı öğrendiği ünlü terimin kökenini biliyorsunuz **segmentation violation (segmentasyon ihlali) veya segmentation fault (segmentasyon hatası).**

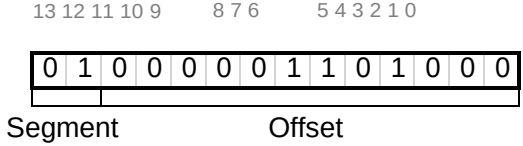
16.2 Hangi Segment'ten (Bölümden) Bahsediyoruz?

Donanım, çeviri sırasında segment kayıtlarını kullanır. Bir segmentteki offseti ve bir adresin hangi segmente başvurduğunu nasıl bilebilir ? Bazen **explicit (açık, belirgin)** bir yaklaşım olarak adlandırılan yaygın bir yaklaşım, adres alanını, sanal adresin en üstteki birkaç bitine göre bölümlere ayırır; bu teknik VAX/VMS sisteminde [LL82] kullanılmıştır. Yukarıdaki örneğimizde üç segmentimiz var; bu yüzden görevimizi yerine getirmek için iki bit'e ihtiyacımız var. Segmenti seçmek için 14 bit sanal adresimizin en üstteki iki bitini kullanırsak , sanal adresimiz şöyle görünür:



Örneğimizde, ilk iki bit 00 ise, donanım sanal adresin kod segmentinde olduğunu bilir ve bu nedenle adresi doğru fiziksel konuma yeniden konumlandırmak için kod tabanını ve sınır çiftini kullanır. En üstteki iki bit 01 ise, donanım adresin yığında olduğunu bilir ,

Ve bu nedenle Heap(Yığın) tabanını ve sınırlanırını kullanır. Örnek Heap(yığın) sanal adresimizi yukarıdan (4200) alalım ve bunun açık olduğundan emin olmak için çevireli. İkili formdaki sanal adres 4200 burada görülebilir:



Resimden de görebileceğiniz gibi, üstteki iki bit (01) donanıma hangi segmentten bahsettiğimizi söyler. En alttaki 12 bit, segmentteki *ofsettir*: 0000 0110 1000 veya onaltılık 0x068 veya ondalık olarak 104. Bu nedenle, donanım hangi segment kaydını kullanılacağını belirlemek için ilk iki biti alır ve ardından sonraki 12 biti segmente offset olarak alır . Temel kaydı ofsete ekleyerek, donanım son fiziksel adrese ulaşır . Ofsetin sınır kontrolünü de kolaylaştırdığını unutmayın: ofsetin sınırlardan daha küçük olup olmadığını kontrol edebiliriz ; değilse, adres illegal'dir. Bu nedenle, taban ve sınırlar diziler olsaydı (segment başına bir girişle), donanım istenen fiziksel adresi elde etmek için böyle bir şey yapıyor olurdu:

```
1 // get top 2 bits of 14-bit VA
2 Segment = (VirtualAddress & SEG_MASK) >> SEG_SHIFT
3 // now get offset
4 Offset  = VirtualAddress & OFFSET_MASK
5 if (Offset >= Bounds[Segment])
6     RaiseException(PROTECTION_FAULT)
7 else
8     PhysAddr = Base[Segment] + Offset
9     Register = AccessMemory(PhysAddr)
```

Çalışan örneğimizde, yukarıdaki sabitler için değerleri doldurabiliriz. Özellikle, SEG MASK 0x3000, SEG SHIFT 12 ve OFFSET_MASK 0xFFF olarak ayarlanacaktır.

Ayrıca, en üstteki iki biti kullandığımızda ve yalnızca üç segmentimiz (kod, yığın, yığın) olduğunda, adres alanının bir bölümünün kullanılmadığını fark etmiş olabilirsiniz. Sanal adres alanını tam olarak kullanmak (ve kullanılmayan bir segmentten kaçınmak) için, bazı sistemler kodu yığınla aynı segmente koyar ve bu nedenle hangi segmentin kullanılacağını seçmek için yalnızca bir bit kullanır [LL82].

Bir segmenti seçmek için çok fazla bit kullanmanın bir başka sorunu da, sanal adres alanının kullanımını sınırlamasıdır. Özellikle, her segment, örneğimizde 4 KB olan *maksimum boyutla* sınırlıdır (segmentleri seçmek için en üstteki iki bitin kullanılması, 16 KB'lık adres alanının dört parçaya veya bu örnekte 4 KB'ye bölündüğü anlamına gelir). Çalışan bir program bir segmenti (Heap veya Stack diyelim) bu maksimumun ötesine büyütmek isterse, programın şansı kalmaz.

Donanımın belirli bir adresin hangi segmentte olduğunu belirlemesinin başka yolları da vardır. **Implicit (örtülü)** yaklaşımda donanım -

adresin nasıl oluştuğunu fark ederek segmenti belirler. Örneğin, adres program sayacından oluşturulduysa (yani, bir komut getirme işlemiydi), adres kod segmenti içindeyse; Adres, Stack veya base pointer(temel işaretçisiyi) temel alıyorsa, Stack segmentinde olmalıdır; diğer tüm adresler Heap de olmalıdır.

16.3 Peki ya Stack (Yığın) ?

Şimdiye kadar, adres alanının önemli bir bileşenini dışarıda bıraktık : Stack. Stack, yukarıdaki diyagramda 28 KB fiziksel adrese taşındı, ancak kritik bir farkla: *geriye doğru büyür* (yani, daha düşük adreslere doğru). Fiziksel bellekte, 28KB[1] 'de "başlar" ve 16KB ve 14KB arasındaki sanal adreslere karşılık gelen 26kb'ye kadar büyür : çeviri farklı şekilde ilerlemelidir.

İhtiyacımız olan ilk şey biraz ekstra donanım desteği. Yalnızca taban ve sınır değerleri yerine , donanımın segmentin hangi yönde büyüdüğünü de bilmesi gerekir (örneğin, segment pozitif yönde büyüdüğünde 1'e ve negatif için 0'a ayarlanmış bir bit). Donanım parçalarının neler olduğuna dair güncellenmiş görünümümüz Şekil 16.4'te görülmektedir:

Parça	Taban	Boyut (maksimum 4K)	Pozitif Büyür mü?
Kod_{00}	32K	2K	1
$Heap_{01}$	34K	3K	1
$Stack_{11}$	28K	2K	0

Şekil 16.4: **Segment Registers(Segment Kayıtları) (Negatif Büyüme Desteği ile)**

Segmentlerin negatif yönde büyüyebileceği yönündeki donanım anlayışıyla, donanım artık bu tür sanal adresleri biraz farklı bir şekilde çevirmesi gerekiyor. İşlemi anlamak için örnek bir Stack (yığın) sanal adresi alıp çevrelim.

Bu örnekte, 27 KB fiziksel adresle eşlenmesi gereken 15 KB sanal adrese erişmek istediğimizi varsayalım. Sanal adresimiz, ikili biçimde, bu nedenle şöyle görünür: 11 1100 0000 0000 (onaltılık 0x3C00).Donanım , segmenti belirlemek için en üstteki iki biti (11) kullanır , ancak daha sonra 3KB'lık bir ofset ile kalırız. Doğru negatif ofseti elde etmek için, maksimum segment boyutunu 3KB'den çıkarmalıyız: bu örnekte, bir segment 4KB olabilir ve bu nedenle doğru negatif ofset 3KB eksi 4KB'dir ve bu da -1KB'ye eşittir. Doğru fiziksel adrese ulaşmak için negatif ofseti (-1KB) tabana(28KB) eklememiz yeterlidir: 27KB. Sınır denetimi, negatif ofsetin mutlak değerinin segmentin geçerli boyutundan küçük veya ona eşit olmasını sağlayarak hesaplanabilir (bu durumda, 2 KB).

[1] Basit olsun diye, stack(yığın) 28KB'de "başlıyor" desek de aslında bu değer geriye doğru büyüyen bölgenin konumunun hemen altındaki bayttır; İlk geçerli bayt aslında 28KB eksi 1'dir. Buna karşılık, ileriye doğru büyüyen bölgeler, segmentin ilk baytının adresinden başlar. Bu yaklaşımı benimsiyoruz çünkü fiziksel adresi hesaplamak için matematiği basit hale getiriyor: fiziksel adres sadece taban artı negatif ofset.

16.4 Paylaşım Desteği

Segmentasyon desteği arttıkça, sistem designer'ları kısa sürede biraz daha fazla donanım desteği ile yeni verimlilik türlerini gerçekleştirebileceklerini fark ettiler. Özellikle, bellekten tasarruf etmek için, bazen belirli bellek kesimlerini adres alanları arasında paylaşmak yararlı olabilir. Özellikle, **code sharing (kod paylaşımı)** yaygındır ve günümüzde sistemlerde hala kullanılmaktadır.

Paylaşımı desteklemek için, donanımdan **koruma bitleri** biçiminde biraz daha fazla desteğe ihtiyacımız var.Temel destek, segment başına birkaç bit ekleyerek, bir programın bir segmenti okuyup yazamayacağını veya belki de segment içinde yer alan kodu çalıştırıp çalıştıramayacağını belirtir.Bir kod segmenti ayarlayarak salt okunur olarak, aynı kod izolasyona zarar verme endişesi olmadan birden fazla işlem arasında paylaşılabilir; Her işlem hala kendi özel belleğini işlediğini düşünürken, işletim sistemi işlem tarafından değiştirilemeyen belleği gizlice paylaşıyor ve böylece illüzyon korunuyor.

Donanım (ve işletim sistemi) tarafından izlenen ek bilgilerin bir örneği Şekil 16.5'te gösterilmiştir. Gördüğünüz gibi, kod segmenti okunacak ve yürütülecek üzere ayarlanmıştır ve böylece bellekteki aynı fiziksel kesim birden çok sanal adres alanına eşlenebilir.

Parça	Taban	Boyut (maksimum 4K)	Pozitif Büyür mü?	Koruma
Kod_{00}	32K	2K	1	Okuma-yürütme
$Heap_{01}$	34K	3K	1	Okuma-Yazma
$Stack_{11}$	28K	2K	0	Okuma-Yazma

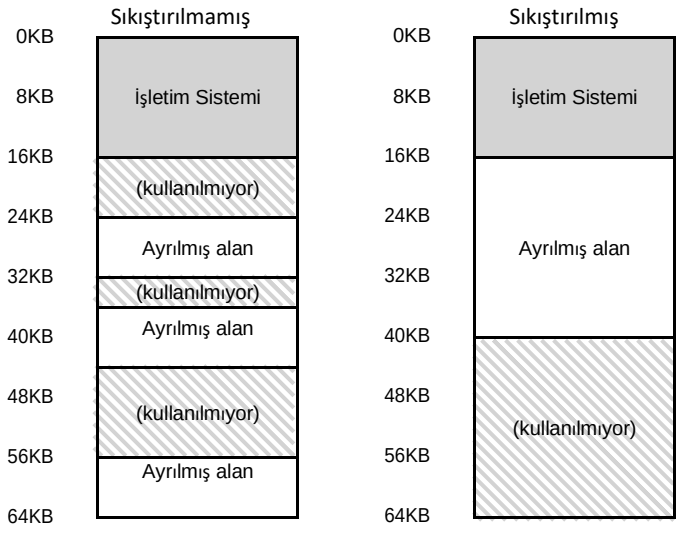
Şekil 16.5: **Segment Register Values (with Protection) Segment Kayıt Değerleri (Korumalı)**

Koruma bitleriyle, daha önce açıklanan donanım algoritmasının da değişmesi gerekecektir. Donanım, sanal bir adresin sınırlar içinde olup olmadığını denetlemenin yanı sıra, belirli bir erişime izin verilip verilmediğini de kontrol etmelidir. Bir kullanıcı işlemi salt okunur bir segmente yazmaya veya yürütülebilir olmayan bir segmentten yürütmeye çalışırsa, donanım bir özel durum oluşturmalı ve böylece işletim sisteminin rahatsız edici işlemle ilgilenmesine izin vermelidir.

16.5 İnci taneli ve kaba taneli Segmentasyon

Şimdiye kadarki örneklerimizin çoğu, yalnızca birkaç segmente (yani kod, Heap, Stack) sahip sistemlere odaklanmıştır; Adres alanını nispeten büyük parçalara ayırdığı için bu segmentasyonu **coarse-grained(kaba taneli)** olarak düşünebiliriz.Kaba parçalar. Bununla birlikte, bazı eski sistemler (örneğin, Multics [CV65, DD 68]) daha esnekti ve adres alanlarının **fine-grained (ince taneli)** segmentasyon olarak adlandırılan çok sayıda küçük segmentten oluşmasına izin verildi.

Birçok segmenti desteklemek, bellekte depolanan bir tür **segment table (segment tablosuyla)** daha da fazla donanım desteği gerektirir. Bu tür segment parçaları genellikle çok sayıda segmentin oluşturulmasını destekler veböylece bir sistemin segmentleri şimdiye kadar tartıştığımızdan daha esnek şekillerde kullanmasını sağlar. Örneğin, Burroughs B5000 gibi ilk makineler binlerce segment için desteğe sahipti ve



Şekil 16.6: **Sıkıştırılmamış ve Sıkıştırılmış Bellek**

Bir derleyicinin kodu ve verileri, işletim sistemi ve donanımın [RK68]. destekleyeceği ayrı segmentlere ayırması bekleniyordu. O zamanki düşünce, ince taneli segmentlere sahip olarak, işletim sisteminin hangi segmentlerin kullanımda olduğunu ve hangilerinin kullanılmadığını daha iyi öğrenebileceği ve böylece ana belleği daha etkili bir şekilde kullanabileceğiydi.

16.6 İşletim Sistemi Desteği

Artık segmentasyonun nasıl çalıştığına dair temel bir fikriniz olmalısınız. Adres alanının parçaları, sistem çalışırken fiziksel belleğe yeniden yerleştirilir ve bu nedenle tüm adres alanı için yalnızca tek bir base/bounds çifti ile daha basit yaklaşımımıza göre büyük bir fiziksel bellek tasarrufu sağlanır. Özellikle, Heap ve Stack arasındaki tüm kullanılmayan tüm alanın fiziksel bellekte ayrılması gerekmez, bu da fiziksel belleğe daha fazla adres alanı sığdırmamıza ve işlem başına büyük ve seyrek bir sanal adres alanını desteklememize olanak tanır.

Bununla birlikte, segmentasyon, işletim sistemi için bir dizi yeni sorunu gündeme getirmektedir . Birincisi eskidir: İşletim sistemi bir bağlam anahtarında ne yapmalı? Şimdiye kadar iyi bir tahminde bulunmalısınız: segment kayıtları kaydedilmeli ve geri yüklenmelidir. Açıkçası, her işlemin kendi sanal adres alanı vardır ve işletim sistemi, işlemin tekrar çalışmasına izin vermeden önce bu kayıtları doğru şekilde ayarladığından emin olmalıdır.

İkincisi, segmentler büyüdüğünde (veya belki de küçüldüğünde) işletim sistemi etkileşimidir. Örneğin, bir program bir nesneyi ayırmak için malloc() öğesini çağırabilir. Bazı durumlarda, mevcut yığın isteğe hizmet verebilir ve böylece

Tavsiye: 1000 Çözüm varsa, büyük bir çözüm yoktur.
Dış parçalanmayı en aza indirmeye çalışan pek çok farklı algoritmanın var olduğu gerçeği, altta yatan daha güçlü bir gerçeğin göstergesidir: Sorunu çözmenin tek bir "en iyi" yolu yoktur. Böylece makul bir şeyle yetinir ve bunun yeterince iyi olduğunu umarız. Tek gerçek çözüm (gelecek bölümlerde göreceğimiz gibi), belleği asla değişken boyutlu parçalara ayırmayarak sorunu tamamen ortadan kaldırmaktır.

malloc() nesne için boş alan bulur ve arayana bir işaretçi döndürür. Ancak diğerlerinde, Heap (yığın) segmentinin kendisinin büyümesi gerekebilir. Bu durumda, bellek ayırma kitaplığı Heap (yığını) büyütmek için bir sistem çağrısı gerçekleştirir (örneğin, geleneksel UNIX sbrk() sistem çağrısı). İşletim sistemi daha sonra (genellikle) daha fazla alan sağlar, segment boyutu kaydını yeni (daha büyük) boyuta günceller ve kütüphaneyi başarı hakkında bilgilendirir; Kütüphane daha sonra yeni nesne için yer ayırabilir ve çağıran programa başarıyla geri dönebilir. İşletim sisteminin, daha fazla fiziksel bellek yoksa veya arama işleminin zaten çok fazla olduğuna karar verirse , isteği reddedebileceğini unutmayın.

Son ve belki de en önemli konu, fiziksel bellekteki boş alanı yönetmektir. Yeni bir adres alanı oluşturulduğunda, işletim sisteminin segmentleri için fiziksel bellekte yer bulabilmesi gerekir. Önceden, her adres alanının aynı boyutta olduğunu varsayıyorduk ve bu nedenle fiziksel bellek, işlemlerin sığacağı bir grup yuva olarak düşünülebilirdi. Şimdi, işlem başına birkaç segmentimiz var ve her segment farklı bir boyutta olabilir.

Ortaya çıkan genel sorun, fiziksel belleğin hızla küçük boş alan delikleriyle dolması, yeni segmentlerin tahsis edilmesini veya mevcut olanları büyütmeyi zorlaştırmasıdır. Bu soruna **external fragmentation (dışsal kırılma)** [R69] diyoruz; bakınız Şekil 16.6 (solda).

Örnekte, bir işlem geliyor ve 20 KB'lık bir segment ayırmak istiyor. Bu örnekte, 24 KB boş alan var, ancak bitişik bir segmentte değil (daha ziyade, bitişik olmayan üç parçada). Bu nedenle, işletim sistemi 20KB isteğini karşılayamaz . Bir segmenti büyütme talebi geldiğinde de benzer sorunlar ortaya çıkabilir; Bir sonraki bu kadar çok baytlık fiziksel alan mevcut değilse, fiziksel belleğin başka bir yerinde boş baytlar olsa bile, işletim sisteminin isteği reddetmesi gerekecektir.

Bu soruna bir çözüm, mevcut segmentleri yeniden düzenleyerek fiziksel belleği **compact(sıkıştırmak)** etmek olacaktır. Örneğin, işletim sistemi hangi işlemlerin çalıştığını durdurabilir, verilerini bitişik bir belleğe kopyalayabilir, segment kayıt değerlerini yeni fiziksel konumlarına işaret edecek şekilde değiştirebilir ve böylece çalışacakları büyük ölçüde boş belleğe sahip olabilir .Bunu yaparak, işletim sistemi yeni ayırma isteğinin başarılı olmasını sağlar. Bununla birlikte, segmentlerin kopyalanması bellek yoğun olduğundan ve genellikle oldukça fazla işlemci süresi kullandığından sıkıştırma pahalıdır;

Sıkıştırılmış fiziksel belleğin diyagramı için bkz. 16.6 (sağda). Sıkıştırma aynı zamanda (ironik bir şekilde) mevcut segmentleri büyütme isteklerini karşılanmasını zorlaştırır ve bu nedenle bu tür talepleri karşılamak için daha fazla yeniden düzenlemeye neden olabilir.

Bunun yerine, daha basit bir yaklaşım, tahsis için büyük miktarda belleği kullanılabilir tutmaya çalışan bir serbest liste yönetim algoritması kullanmak olabilir. **best-fit** gibi (boş alanların listesini tutan) klasik algoritmalar da dahil olmak üzere insanların benimsediği yüzlerce yaklaşım vardır ve istekte bulunan kişiye istenen ayırmayı karşılayan boyut olarak en yakın olanı döndürür **worst-fit**, **first-fit** ve **buddy algorithm** [K68] gibi daha karmaşık şemalar. Wilson ve diğerleri tarafından yapılan mükemmel bir anket, bu tür algoritmalar [W + 95] hakkında daha fazla bilgi edinmek istiyorsanız başlamak için iyi bir yerdir veya daha sonraki bir bölümde bazı temel bilgileri ele alana kadar bekleyebilirsiniz. Ne yazık ki, algoritma ne kadar akıllı olursa olsun, dış parçalanma olacaktır. hala var; Bu nedenle, iyi bir algoritma basitçe onu en aza indirmeye çalışır.

16.7 Özet

Segmentasyon bir dizi sorunu çözer ve belleğin daha etkili bir sanallaştırmasını oluşturmamıza yardımcı olur. Yalnızca dinamik yer değiştirmenin ötesinde, adresin alanının mantıksal bölümleri arasındaki büyük potansiyel bellek israfını önleyerek seyrek adres alanlarını daha iyi destekleyebilir. Aritmetik bölümlemenin gerektirdiği işlemleri yapmak kolay ve donanıma çok uygun olduğu için hızlıdır ; çevirinin genel giderleri minimumdur. Bir yan fayda da ortaya çıkıyor: kod paylaşımı. Kod ayrı bir segmente yerleştirilirse , böyle bir segment potansiyel olarak birden çok çalışan program arasında paylaşılabilir.

Ancak, öğrendiğimiz gibi, belleğe değişken boyutlu segmentler ayırmak, üstesinden gelmek istediğimiz bazı sorunlara yol açmaktadır. Birincisi, yukarıda tartışıldığı gibi, dışsal parçalanmadır. Segmentler değişken boyutlu olduğundan, boş bellek tek boyutlu parçalara bölünür ve bu nedenle bir bellek ayırma isteğini yerine getirmek zor olabilir. Akıllı algoritmalar [W+95] veya periyodik olarak kompakt bellek kullanılmaya çalışılabilir,ancak sorun temeldir ve kaçınılması gereken bir durumdur.

İkinci ve belki de daha önemli sorun, segmentasyonun tamamen genelleştirilmiş, seyrek adres alanımızı destekleyecek kadar esnek olmamasıdır. Örneğin, tümü tek bir mantıksal segmentte buluunan büyük ama seyrek kullanılan bir Heap(yığınımız) varsa, erişilebilmesi için tüm Heap(yığının) bellekte kalması gerekir . Başka bir deyişle, adres alanının nasıl kullanıldığına dair modelimiz, altta yatan segmentasyonun onu desteklemek için nasıl tasarlandığıyla tam olarak eşleşmiyorsa, segmentasyon çok iyi çalışmaz. Bu nedenle bazı yeni çözümler bulmamız gerekiyor. Onları bulmaya hazır mısınız?

References

[CV65] "Introduction and Overview of the Multics System" by F. J. Corbato, V. A. Vyssotsky. Fall Joint Computer Conference, 1965. *One of five papers presented on Multics at the Fall Joint Computer Conference; oh to be a fly on the wall in that room that day!*

[DD68] "Virtual Memory, Processes, and Sharing in Multics" by Robert C. Daley and Jack B. Dennis. Communications of the ACM, Volume 11:5, May 1968. *An early paper on how to perform dynamic linking in Multics, which was way ahead of its time. Dynamic linking finally found its way back into systems about 20 years later, as the large X-windows libraries demanded it. Some say that these large X11 libraries were MIT's revenge for removing support for dynamic linking in early versions of* UNIX*!*

[G62] "Fact Segmentation" by M. N. Greenfield. Proceedings of the SJCC, Volume 21, May 1962. *Another early paper on segmentation; so early that it has no references to other work.*

[H61] "Program Organization and Record Keeping for Dynamic Storage" by A. W. Holt. Com- munications of the ACM, Volume 4:10, October 1961. *An incredibly early and difficult to read paper about segmentation and some of its uses.*

[I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals" by Intel. 2009. Avail- able: http://www.intel.com/products/processor/manuals. *Try reading about segmentation in here (Chapter 3 in Volume 3a); it'll hurt your head, at least a little bit.*

[K68] "The Art of Computer Programming: Volume I" by Donald Knuth. Addison-Wesley, 1968. *Knuth is famous not only for his early books on the Art of Computer Programming but for his typesetting system TeX which is still a powerhouse typesetting tool used by professionals today, and indeed to typeset this very book. His tomes on algorithms are a great early reference to many of the algorithms that underly computing systems today.*

[L83] "Hints for Computer Systems Design" by Butler Lampson. ACM Operating Systems Review, 15:5, October 1983. *A treasure-trove of sage advice on how to build systems. Hard to read in one sitting; take it in a little at a time, like a fine wine, or a reference manual.*

[LL82] "Virtual Memory Management in the VAX/VMS Operating System" by Henry M. Levy, Peter H. Lipman. IEEE Computer, Volume 15:3, March 1982. *A classic memory management system, with lots of common sense in its design. We'll study it in more detail in a later chapter.*

[RK68] "Dynamic Storage Allocation Systems" by B. Randell and C.J. Kuehner. Communica- tions of the ACM, Volume 11:5, May 1968. *A nice overview of the differences between paging and segmentation, with some historical discussion of various machines.*

[R69] "A note on storage fragmentation and program segmentation" by Brian Randell. Com- munications of the ACM, Volume 12:7, July 1969. *One of the earliest papers to discuss fragmenta- tion.*

[W+95] "Dynamic Storage Allocation: A Survey and Critical Review" by Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles. International Workshop on Memory Management, Scotland, UK, September 1995. *A great survey paper on memory allocators.*

Ödev (Simülasyon)

Bu program, segmentasyonlu bir sistemde adres çevirilerinin nasıl yapıldığını görmenizi sağlar. Ayrıntılar için README'ye bakın.

Sorular

1. İlk önce bazı adresleri çevirmek için küçük bir adres alanı kullanalım. İşte birkaç farklı rastgele içeren basit bir parametre kümesi; adresleri çevirebilir misiniz?

```
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512
  -L 20 -s 0
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512
  -L 20 -s 1
segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512
  -L 20 -s 2
```

1- Ilk kodumuzu (segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0) Ubuntu'da çevirdiğimizde karşımıza çıkan ekran şu şekilde olacaktır:

```
emre@emre-virtual-machine:~/Desktop/ostep/ostep-homework/vm-segmentation$ python3 segmentation.py
-a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0
ARG seed 0
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20

  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20

Virtual Address Trace
  VA  0: 0x0000006c (decimal:  108) --> PA or segmentation violation?
  VA  1: 0x00000061 (decimal:   97) --> PA or segmentation violation?
  VA  2: 0x00000035 (decimal:   53) --> PA or segmentation violation?
  VA  3: 0x00000021 (decimal:   33) --> PA or segmentation violation?
  VA  4: 0x00000041 (decimal:   65) --> 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 address space with two segments: the top
bit of the virtual address can thus be used to check whether the virtual address
is in segment 0 (topbit=0) or segment 1 (topbit=1). Note that the base/limit pairs
given to you grow in different directions, depending on the segment, i.e., segment 0
grows in the positive direction, whereas segment 1 in the negative.
```

Burada sanal adreslerimizin hangi segmentasyon bölgesinde olduğunu ve geçerli olup olmadığını kontrol etmek için -c flag (bayrağıyla) ile kontrol ediyoruz. Onun çıktısıda şu şekilde olacaktır:

```
emre@emre-virtual-machine:~/Desktop/ostep/ostep-homework/vm-segmentation$ python3 segmentation.py
 -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0 -c
ARG seed 0
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20

  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20

Virtual Address Trace
  VA  0: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
  VA  1: 0x00000061 (decimal:   97) --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0x00000035 (decimal:   53) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000021 (decimal:   33) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000041 (decimal:   65) --> SEGMENTATION VIOLATION (SEG1)
```

Her bir satırda verilen, bir sanal adres (VA) belirtilir ve bu adresin geçerli bir bellek bölgesine (seg) işaret edip etmediğini belirtilir. Örneğin, ilk satırda, VA 0 olarak belirtilen 0x0000006c adresi SEG1 bölgesinde geçerli bir adres olarak işaretlenmiştir. (0x0000001ec olarak değiştirilmiştir). Ancak diğer satırlarda belirtilen hiçbiri geçerli bir bellek bölgesine işaret etmediğinden SEGMENTATION VIOLATION (segmentasyon ihlali) hatası oluşmuştur.

(segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1) kodumuzu Ubuntu terminalde çevirdiğimizde karşımıza çıkan ekran şı şekildedir :

```
emre@emre-virtual-machine:~/Desktop/ostep/ostep-homework/vm-segmentation$ python3 segmentation.py
-a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1
ARG seed 1
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20

  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20

Virtual Address Trace
  VA  0: 0x00000011 (decimal:   17) --> PA or segmentation violation?
  VA  1: 0x0000006c (decimal:  108) --> PA or segmentation violation?
  VA  2: 0x00000061 (decimal:   97) --> PA or segmentation violation?
  VA  3: 0x00000020 (decimal:   32) --> PA or segmentation violation?
  VA  4: 0x0000003f (decimal:   63) --> 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 address space with two segments: the top
bit of the virtual address can thus be used to check whether the virtual address
is in segment 0 (topbit=0) or segment 1 (topbit=1). Note that the base/limit pairs
given to you grow in different directions, depending on the segment, i.e., segment 0
grows in the positive direction, whereas segment 1 in the negative.
```

Tekrar -c flag'i (bayrağı) ile control edersek çıktımız şu şekildedir :

```
emre@emre-virtual-machine:~/Desktop/ostep/ostep-homework/vm-segmentation$ python3 segmentation.py
-a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 1 -c
ARG seed 1
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20

  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20

Virtual Address Trace
  VA  0: 0x00000011 (decimal:   17) --> VALID in SEG0: 0x00000011 (decimal:   17)
  VA  1: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
  VA  2: 0x00000061 (decimal:   97) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x00000020 (decimal:   32) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x0000003f (decimal:   63) --> SEGMENTATION VIOLATION (SEG0)
```

İlk iki satırda, VA 0 ve VA 1 olarak belirtilen 0x00000011 ve 0x0000006c adresleri SEG0 VE SEG1 bölgelerinde geçerli adresler olarak işaretlenmiştir. (0x00000011 ve 0x000001ec olarak değiştirilmiştir). Ancak diğer satırlarda belirtilen adrselerin hiçbiri geçerli bir bellek bölgesine işaret etmediğinden, SEGMENTATION VIOLATION
(segmentasyon ihlali) hatası oluşmuştur.

(segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2) kodumuzu Ubuntu terminalde çevirdiğimizde karşımıza çıkan ekran şı şekildedir:

```
emre@emre-virtual-machine:~/Desktop/ostep/ostep-homework/vm-segmentation$ python3 segmentation.py
-a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2
ARG seed 2
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20

  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20

Virtual Address Trace
  VA  0: 0x0000007a (decimal:  122) --> PA or segmentation violation?
  VA  1: 0x00000079 (decimal:  121) --> PA or segmentation violation?
  VA  2: 0x00000007 (decimal:    7) --> PA or segmentation violation?
  VA  3: 0x0000000a (decimal:   10) --> PA or segmentation violation?
  VA  4: 0x0000006a (decimal:  106) --> 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 address space with two segments: the top
bit of the virtual address can thus be used to check whether the virtual address
is in segment 0 (topbit=0) or segment 1 (topbit=1). Note that the base/limit pairs
given to you grow in different directions, depending on the segment, i.e., segment 0
grows in the positive direction, whereas segment 1 in the negative.
```

Tekrar -c flag'i (bayrağı) ile control edersek çıktımız şu şekildedir :

```
emre@emre-virtual-machine:~/Desktop/ostep/ostep-homework/vm-segmentation$ python3 segmentation.py
-a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 2 -c
ARG seed 2
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20

  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20

Virtual Address Trace
  VA  0: 0x0000007a (decimal:  122) --> VALID in SEG1: 0x000001fa (decimal:  506)
  VA  1: 0x00000079 (decimal:  121) --> VALID in SEG1: 0x000001f9 (decimal:  505)
  VA  2: 0x00000007 (decimal:    7) --> VALID in SEG0: 0x00000007 (decimal:    7)
  VA  3: 0x0000000a (decimal:   10) --> VALID in SEG0: 0x0000000a (decimal:   10)
  VA  4: 0x0000006a (decimal:  106) --> SEGMENTATION VIOLATION (SEG1)
```

İlk satırda VA 0 olarak belirtilen 0x0000007a adresi SEG1 bölgesinde geçerli adres olarak işaretlenmiştir. (0x000001fa olarak değiştirilmiştir)

VA 1 olarak belirtilen 0x00000079 adresi SEG1 bölgesinde geçerli adres olarak işaretlenmiştir. (0x000001f9 olarak değiştirilmiştir)

VA 2 olarak belirtilen 0x00000007 adresi SEG0 bölgesinde geçerli adres olarak işaretlenmiştir. (0x0000007 olarak değiştirilmiştir)

VA 3 olarak belirtilen 0x0000000a adresi SEG0 bölgesinde geçerli adres olarak işaretlenmiştir. (0x0000000a olarak değiştirilmiştir)

Ancak son satırda VA 4 olarak belirtilen 0x0000006a adresi geçerli bir bellek bölgesi işaret edilmediğinden, SEGMENTATION VIOLATION (segmentasyon ihlali) hatası oluşmuştur.

2. Şimdi, yapılandırdığımız bu küçük adres alanını anlayıp anlamadığımızı görelim (yukarıdaki sorudaki parametreleri kullanarak). Segment 0'daki en yüksek yasal sanal adres nedir ? Peki ya segment 1'deki en düşük yasal sanal adres? Tüm bu adres alanındaki en düşük ve en yüksek *yasadışı* adresler nelerdir? Son olarak, haklı olup olmadığınızı test etmek için -A bayrağı ile segmentation.py nasıl çalıştırırsınız ?

Segment 0'daki en yüksek sanal adres 0x00000011 (decimal: 17) iken, segment 1'deki en düşük yasal sanal adres 0x0000006c (decimal: 108) olmuştur. Tüm bu adres alanındaki en düşük yasa dışı adres 0x00000020(decimal:32) iken en yüksek yasadışı adres ise 0x00000061(decimal: 97) olmuştur.

Yukarıdaki kodların çıktısını şu şekilde alarak control edebiliriz :

python3 segmentation.py -A 17,108,32,97 -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c

Kod çıktımızda bu şekilde oluyor :

```
emre@emre-virtual-machine:~/Desktop/ostep/ostep-homework/vm-segmentation$ python3 segmentation.py
 -A 17,108,32,97 -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -c
ARG seed 0
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 20

  Segment 1 base  (grows negative) : 0x00000200 (decimal 512)
  Segment 1 limit                  : 20

Virtual Address Trace
  VA  0: 0x00000011 (decimal:   17) --> VALID in SEG0: 0x00000011 (decimal:   17)
  VA  1: 0x0000006c (decimal:  108) --> VALID in SEG1: 0x000001ec (decimal:  492)
  VA  2: 0x00000020 (decimal:   32) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000061 (decimal:   97) --> SEGMENTATION VIOLATION (SEG1)
```

3. Diyelim ki 128 baytlık bir fiziksel bellekte 16 baytlık küçük bir adres alanımız var. Simülatörün belirtilen adres akışı içim aşağıdaki çeviri sonuçlarını oluşturmasını sağlamak için hangi taban ve sınırları ayarlarsınız: geçerli, geçerli, ihlal, ….,ihlal, geçerli, geçerli şekildemi ? Aşağıdaki parametreleri varsayalım:

```
segmentation.py -a 16 -p 128
  -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
  --b0 ? --l0 ? --b1 ? --l1 ?
```

İstenen çeviri sonuçlarını elde etmek için segmentasyon sistemini kullanarak iki farklı taban ve sınır değerlerine sahip iki segment ayarlamamız gerekir. İlk segmentin, adres akışının ilk iki adresini (0 ve 1) içeren sınırları içinde olmasını, diğer adreslerin ise sınırların dışında kalmasını sağlamamız gerekmektedir.-

İkinci segmentin ise adres akışının son iki adresini (14 ve 15)içeren sınırları içinde olmasını, diğer adreslerin ise sınırların dışında kalmasını sağlamamız gerekmektedir.

Bu yüzden, ilk segmentin taban ve sınır değerlerini 0 ve 2 olarak ayarlayabiliriz.Bu, ilk segmentin adres akışının ilk iki adresini (0 ve 1) kapsayacağı anlamına gelecek ve ilk iki çevirinin "geçerli" olacağı anlamına gelmektedir. İkinci segment için ise taban ve sınır değerlerini 14 ve 2 olarak ayarlayabiliriz. Bu, ikinci segmentin adres akışının son iki adresini (14 ve 15) kapsayacağı anlamına gelecek ve son iki çevirinin "geçerli" olacağı anlamına gelecektir. Diğer tüm adresler ise iki segmentin sınırlarının dışında kalacak ve " ihlal "çevirilerine neden olacaktir.

Ubuntu terminalinde kullanacağımız kod şekildedir:
python3 segmentation.py -a 16 -p 128 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 --b0 0 --l0 2 --b1 14 --l1 2 -c

```
emre@emre-virtual-machine:~/Desktop/ostep/ostep-homework/vm-segmentation$ python3 segmentation.py
 -a 16 -p 128 -A 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 --b0 0 --l0 2 --b1 14 --l1 2 -c
ARG seed 0
ARG address space size 16
ARG phys mem size 128

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 2

  Segment 1 base  (grows negative) : 0x0000000e (decimal 14)
  Segment 1 limit                  : 2

Virtual Address Trace
  VA  0: 0x00000000 (decimal:    0) --> VALID in SEG0: 0x00000000 (decimal:    0)
  VA  1: 0x00000001 (decimal:    1) --> VALID in SEG0: 0x00000001 (decimal:    1)
  VA  2: 0x00000002 (decimal:    2) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000003 (decimal:    3) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000004 (decimal:    4) --> SEGMENTATION VIOLATION (SEG0)
  VA  5: 0x00000005 (decimal:    5) --> SEGMENTATION VIOLATION (SEG0)
  VA  6: 0x00000006 (decimal:    6) --> SEGMENTATION VIOLATION (SEG0)
  VA  7: 0x00000007 (decimal:    7) --> SEGMENTATION VIOLATION (SEG0)
  VA  8: 0x00000008 (decimal:    8) --> SEGMENTATION VIOLATION (SEG1)
  VA  9: 0x00000009 (decimal:    9) --> SEGMENTATION VIOLATION (SEG1)
  VA 10: 0x0000000a (decimal:   10) --> SEGMENTATION VIOLATION (SEG1)
  VA 11: 0x0000000b (decimal:   11) --> SEGMENTATION VIOLATION (SEG1)
  VA 12: 0x0000000c (decimal:   12) --> SEGMENTATION VIOLATION (SEG1)
  VA 13: 0x0000000d (decimal:   13) --> SEGMENTATION VIOLATION (SEG1)
  VA 14: 0x0000000e (decimal:   14) --> VALID in SEG1: 0x0000000c (decimal:   12)
  VA 15: 0x0000000f (decimal:   15) --> VALID in SEG1: 0x0000000d (decimal:   13)
```

4. Rastgele oluşturulmuş sanal adreslerin yaklaşık %90'ının geçerli olduğu bir sorun oluşturmak istediğimizi varsayalım (segmentasyon ihlalleri değil). Bunu yapmak için simülatörü nasıl yapılandırmalısınız ? Bu sonucu elde etmek için hangi parametreler önemlidir ?

   Bu sonucu elde etmek taban ve sınır değerlerini ayarlamamız gerekir. Bunu yapmak için temel değerini adres alanının büyüklüğünün yarısından daha büyük, ancak sınır değerini adres alanının büyüklüğünden daha küçük bir degree ayarlayabiliriz.Bu, adres akışının çoğunun segment içinde olacağı anlamına gelecek ve "geçerli" çevirilerinin %90'ının elde edileceği anlamına gelecektir.

   Örneğin, adres alanının 16 bayt olduğunu ve rastgele oluşturulmuş adresleri kullandığımızı varsayalım.Bu durumda taban değerini 8 ve sınır değerini 7 olarak ayarlayabiliriz. Bu, adres akışının çoğunun segment içinde olacağı anlamına gelcek ve " geçerli " çevirilerinin %90'ının elde edileceğ anlamına gelecektir.

   Sonucu elde etmek için öneli olan parametreler taban ve sınır değerleridir. Taban değeri adres alanının büyüklüğünün yarısından daha büyük, ancak sınır değerini adres alanının büyüklüğünden daha küçük bir değere ayarlamak gerekir.

   Örnek olarak oluşturduğumuz kodumuz şu şekildedir:

   python3 segmentation.py -a 16 -p 128 -n 90 -b 8 -l 7 -c

   kod çıktımız bir sonraki sayfa gösterilmiştir :

```
emre@emre-virtual-machine:~/Desktop/ostep/ostep-homework/vm-segmentation$ python3 segmentation.py
 -a 16 -p 128 -n 90 -b 8 -l 7 -c
ARG seed 0
ARG address space size 16
ARG phys mem size 128

Segment register information:

  Segment 0 base  (grows positive) : 0x00000008 (decimal 8)
  Segment 0 limit                  : 7

  Segment 1 base  (grows negative) : 0x00000068 (decimal 104)
  Segment 1 limit                  : 7

Virtual Address Trace
  VA  0: 0x00000006 (decimal:     6) --> VALID in SEG0: 0x0000000e (decimal:    14)
  VA  1: 0x00000004 (decimal:     4) --> VALID in SEG0: 0x0000000c (decimal:    12)
  VA  2: 0x00000008 (decimal:     8) --> SEGMENTATION VIOLATION (SEG1)
  VA  3: 0x00000006 (decimal:     6) --> VALID in SEG0: 0x0000000e (decimal:    14)
  VA  4: 0x0000000c (decimal:    12) --> VALID in SEG1: 0x00000064 (decimal:   100)
  VA  5: 0x00000004 (decimal:     4) --> VALID in SEG0: 0x0000000c (decimal:    12)
  VA  6: 0x00000007 (decimal:     7) --> SEGMENTATION VIOLATION (SEG0)
  VA  7: 0x00000009 (decimal:     9) --> VALID in SEG1: 0x00000061 (decimal:    97)
  VA  8: 0x0000000e (decimal:    14) --> VALID in SEG1: 0x00000066 (decimal:   102)
  VA  9: 0x00000008 (decimal:     8) --> SEGMENTATION VIOLATION (SEG1)
  VA 10: 0x00000004 (decimal:     4) --> VALID in SEG0: 0x0000000c (decimal:    12)
  VA 11: 0x0000000c (decimal:    12) --> VALID in SEG1: 0x00000064 (decimal:   100)
  VA 12: 0x00000009 (decimal:     9) --> VALID in SEG1: 0x00000061 (decimal:    97)
  VA 13: 0x00000004 (decimal:     4) --> VALID in SEG0: 0x0000000c (decimal:    12)
  VA 14: 0x0000000e (decimal:    14) --> VALID in SEG1: 0x00000066 (decimal:   102)
  VA 15: 0x0000000f (decimal:    15) --> VALID in SEG1: 0x00000067 (decimal:   103)
  VA 16: 0x0000000c (decimal:    12) --> VALID in SEG1: 0x00000064 (decimal:   100)
  VA 17: 0x0000000e (decimal:    14) --> VALID in SEG1: 0x00000066 (decimal:   102)
  VA 18: 0x00000004 (decimal:     4) --> VALID in SEG0: 0x0000000c (decimal:    12)
  VA 19: 0x0000000b (decimal:    11) --> VALID in SEG1: 0x00000063 (decimal:    99)
  VA 20: 0x0000000e (decimal:    14) --> VALID in SEG1: 0x00000066 (decimal:   102)
  VA 21: 0x0000000a (decimal:    10) --> VALID in SEG1: 0x00000062 (decimal:    98)
  VA 22: 0x00000007 (decimal:     7) --> SEGMENTATION VIOLATION (SEG0)
  VA 23: 0x00000001 (decimal:     1) --> VALID in SEG0: 0x00000009 (decimal:     9)
  VA 24: 0x00000006 (decimal:     6) --> VALID in SEG0: 0x0000000e (decimal:    14)
  VA 25: 0x00000009 (decimal:     9) --> VALID in SEG1: 0x00000061 (decimal:    97)
  VA 26: 0x0000000e (decimal:    14) --> VALID in SEG1: 0x00000066 (decimal:   102)
  VA 27: 0x0000000f (decimal:    15) --> VALID in SEG1: 0x00000067 (decimal:   103)
  VA 28: 0x00000007 (decimal:     7) --> SEGMENTATION VIOLATION (SEG0)
  VA 29: 0x0000000d (decimal:    13) --> VALID in SEG1: 0x00000065 (decimal:   101)
  VA 30: 0x00000004 (decimal:     4) --> VALID in SEG0: 0x0000000c (decimal:    12)
  VA 31: 0x0000000c (decimal:    12) --> VALID in SEG1: 0x00000064 (decimal:   100)
  VA 32: 0x00000008 (decimal:     8) --> SEGMENTATION VIOLATION (SEG1)
  VA 33: 0x00000000 (decimal:     0) --> VALID in SEG0: 0x00000008 (decimal:     8)
  VA 34: 0x0000000b (decimal:    11) --> VALID in SEG1: 0x00000063 (decimal:    99)
  VA 35: 0x00000006 (decimal:     6) --> VALID in SEG0: 0x0000000e (decimal:    14)
  VA 36: 0x0000000d (decimal:    13) --> VALID in SEG1: 0x00000065 (decimal:   101)
  VA 37: 0x0000000a (decimal:    10) --> VALID in SEG1: 0x00000062 (decimal:    98)
  VA 38: 0x00000000 (decimal:     0) --> VALID in SEG0: 0x00000008 (decimal:     8)
  VA 39: 0x00000007 (decimal:     7) --> SEGMENTATION VIOLATION (SEG0)
  VA 40: 0x0000000d (decimal:    13) --> VALID in SEG1: 0x00000065 (decimal:   101)
  VA 41: 0x00000003 (decimal:     3) --> VALID in SEG0: 0x0000000b (decimal:    11)
  VA 42: 0x00000005 (decimal:     5) --> VALID in SEG0: 0x0000000d (decimal:    13)
  VA 43: 0x0000000d (decimal:    13) --> VALID in SEG1: 0x00000065 (decimal:   101)
  VA 44: 0x00000003 (decimal:     3) --> VALID in SEG0: 0x0000000b (decimal:    11)
  VA 45: 0x00000009 (decimal:     9) --> VALID in SEG1: 0x00000061 (decimal:    97)
  VA 46: 0x00000003 (decimal:     3) --> VALID in SEG0: 0x0000000b (decimal:    11)
  VA 47: 0x0000000f (decimal:    15) --> VALID in SEG1: 0x00000067 (decimal:   103)
  VA 48: 0x0000000c (decimal:    12) --> VALID in SEG1: 0x00000064 (decimal:   100)
  VA 49: 0x00000007 (decimal:     7) --> SEGMENTATION VIOLATION (SEG0)
  VA 50: 0x00000001 (decimal:     1) --> VALID in SEG0: 0x00000009 (decimal:     9)
  VA 51: 0x00000005 (decimal:     5) --> VALID in SEG0: 0x0000000d (decimal:    13)
  VA 52: 0x00000008 (decimal:     8) --> SEGMENTATION VIOLATION (SEG1)
  VA 53: 0x0000000e (decimal:    14) --> VALID in SEG1: 0x00000066 (decimal:   102)
  VA 54: 0x00000001 (decimal:     1) --> VALID in SEG0: 0x00000009 (decimal:     9)
  VA 55: 0x00000008 (decimal:     8) --> SEGMENTATION VIOLATION (SEG1)
  VA 56: 0x0000000b (decimal:    11) --> VALID in SEG1: 0x00000063 (decimal:    99)
  VA 57: 0x00000008 (decimal:     8) --> SEGMENTATION VIOLATION (SEG1)
  VA 58: 0x0000000d (decimal:    13) --> VALID in SEG1: 0x00000065 (decimal:   101)
  VA 59: 0x00000008 (decimal:     8) --> SEGMENTATION VIOLATION (SEG1)
  VA 60: 0x0000000f (decimal:    15) --> VALID in SEG1: 0x00000067 (decimal:   103)
  VA 61: 0x00000009 (decimal:     9) --> VALID in SEG1: 0x00000061 (decimal:    97)
  VA 62: 0x00000009 (decimal:     9) --> VALID in SEG1: 0x00000061 (decimal:    97)
  VA 63: 0x00000007 (decimal:     7) --> SEGMENTATION VIOLATION (SEG0)
  VA 64: 0x00000009 (decimal:     9) --> VALID in SEG1: 0x00000061 (decimal:    97)
  VA 65: 0x00000006 (decimal:     6) --> VALID in SEG0: 0x0000000e (decimal:    14)
  VA 66: 0x00000009 (decimal:     9) --> VALID in SEG1: 0x00000061 (decimal:    97)
  VA 67: 0x00000004 (decimal:     4) --> VALID in SEG0: 0x0000000c (decimal:    12)
  VA 68: 0x00000003 (decimal:     3) --> VALID in SEG0: 0x0000000b (decimal:    11)
  VA 69: 0x00000002 (decimal:     2) --> VALID in SEG0: 0x0000000a (decimal:    10)
  VA 70: 0x00000009 (decimal:     9) --> VALID in SEG1: 0x00000061 (decimal:    97)
  VA 71: 0x0000000a (decimal:    10) --> VALID in SEG1: 0x00000062 (decimal:    98)
  VA 72: 0x00000007 (decimal:     7) --> SEGMENTATION VIOLATION (SEG0)
  VA 73: 0x00000001 (decimal:     1) --> VALID in SEG0: 0x00000009 (decimal:     9)
  VA 74: 0x0000000c (decimal:    12) --> VALID in SEG1: 0x00000064 (decimal:   100)
  VA 75: 0x0000000e (decimal:    14) --> VALID in SEG1: 0x00000066 (decimal:   102)
  VA 76: 0x0000000e (decimal:    14) --> VALID in SEG1: 0x00000066 (decimal:   102)
  VA 77: 0x0000000d (decimal:    13) --> VALID in SEG1: 0x00000065 (decimal:   101)
  VA 78: 0x0000000e (decimal:    14) --> VALID in SEG1: 0x00000066 (decimal:   102)
  VA 79: 0x0000000e (decimal:    14) --> VALID in SEG1: 0x00000066 (decimal:   102)
  VA 80: 0x00000008 (decimal:     8) --> SEGMENTATION VIOLATION (SEG1)
  VA 81: 0x00000006 (decimal:     6) --> VALID in SEG0: 0x0000000e (decimal:    14)
  VA 82: 0x0000000b (decimal:    11) --> VALID in SEG1: 0x00000063 (decimal:    99)
  VA 83: 0x00000004 (decimal:     4) --> VALID in SEG0: 0x0000000c (decimal:    12)
  VA 84: 0x0000000c (decimal:    12) --> VALID in SEG1: 0x00000064 (decimal:   100)
  VA 85: 0x0000000d (decimal:    13) --> VALID in SEG1: 0x00000065 (decimal:   101)
  VA 86: 0x0000000e (decimal:    14) --> VALID in SEG1: 0x00000066 (decimal:   102)
  VA 87: 0x00000009 (decimal:     9) --> VALID in SEG1: 0x00000061 (decimal:    97)
  VA 88: 0x0000000f (decimal:    15) --> VALID in SEG1: 0x00000067 (decimal:   103)
  VA 89: 0x00000009 (decimal:     9) --> VALID in SEG1: 0x00000061 (decimal:    97)
```

5. Simülatörü hiçbir sanal adres geçerli olmayacak şekilde çalıştırabilir misiniz? Nasıl?

Simülatörü hiçbir sanal adres geçerli olmayacak şekilde çalıştırmak için, segment tabanı ve sınırlarını ayarlamak gerekir. Örneğin, aşağıdaki komut satırı seçenekleri segmen 0 tabanını 0 ve sınırını 0, segment 1 tabanını 8 ve sınırını 0 yapar:

python3 segmentation.py -a 16 -p 128 --b0 0 --l0 0 --b1 8 --l1 0 -c

Bu ayarlarla, simülator çalıştırıldığında, hiçbir sanal adres geçerli olmayacak ve hiçbir çeviri gerçekleştirilemeyecektir. Kod çıktısı şu şekilde olacaktır :

```
emre@emre-virtual-machine:~/Desktop/ostep/ostep-homework/vm-segmentation$ python3 segmentation.py
 -a 16 -p 128 --b0 0 --l0 0 --b1 8 --l1 0 -c
ARG seed 0
ARG address space size 16
ARG phys mem size 128

Segment register information:

  Segment 0 base  (grows positive) : 0x00000000 (decimal 0)
  Segment 0 limit                  : 0

  Segment 1 base  (grows negative) : 0x00000008 (decimal 8)
  Segment 1 limit                  : 0

Virtual Address Trace
  VA  0: 0x0000000d (decimal:   13) --> SEGMENTATION VIOLATION (SEG1)
  VA  1: 0x0000000c (decimal:   12) --> SEGMENTATION VIOLATION (SEG1)
  VA  2: 0x00000006 (decimal:    6) --> SEGMENTATION VIOLATION (SEG0)
  VA  3: 0x00000004 (decimal:    4) --> SEGMENTATION VIOLATION (SEG0)
  VA  4: 0x00000008 (decimal:    8) --> SEGMENTATION VIOLATION (SEG1)
```