16

Segmentasyon

Şu ana kadar, her işlem için tüm adres alanını bellekte tutuyorduk. Temel ve sınır kayıtları sayesinde işletim sistemi(OS), işlemleri fiziksel belleğin farklı bölümlerine kolayca yeniden yerleştirebilir. Ancak, bu adres alanlarımızda ilginç bir şey fark etmiş olabilirsiniz: **stack** ve **heap** arasında büyük bir boşluk(free) var.

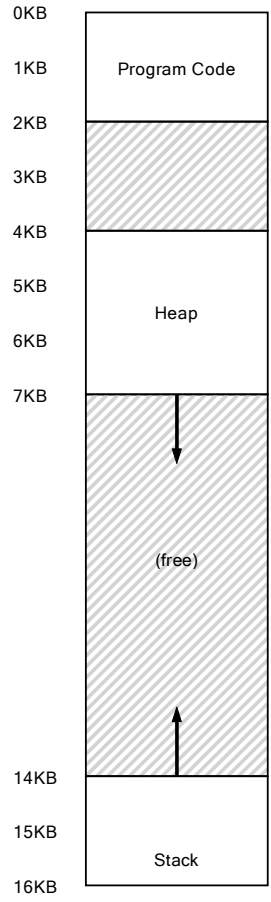
16.1 numaralı resimden anlayabileceğiniz gibi, işlem tarafından kullanılmayan **heap** ve **stack** arasındaki boşluk hâlâ fiziksel bellekte yer kaplıyor. Bu nedenle, belleği sanallaştırmak için temel ve sınır kayıt çiftini kullanan basit yöntem israf edicidir. Ayrıca, tüm adres alanının bellekte sığmadığında bir programı çalıştırmak da oldukça zordur; bu nedenle, temel ve sınırlar istediğimiz kadar esnek değildir. Ve böylece:

KRİTİK KONU: GENİŞ BİR ADRES ALANI NASIL DESTEKLENİR?

Stack ve heap arasında (potansiyel olarak) çok fazla boş alan bulunan büyük bir adres alanını nasıl destekleyebiliriz? Küçük (sahte) adres alanları içeren örneklerimizde israfın çok da 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 Temel/Sınırlar

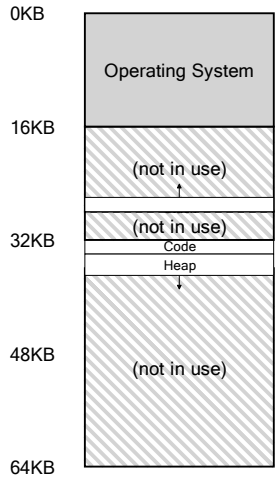
Bu sorunu çözmek için bir fikir doğdu ve buna **segmentasyon(segmentation)** deniyor. Bu, en azından 1960'ların [H61, G62] başlarına kadar uzanan oldukça eski bir fikirdir. Fikir basit: MMU'muzda yalnızca bir taban ve sınır çiftine sahip olmak yerine, neden adres uzayının mantıksal **segmenti** başına bir taban ve sınır çifti olmasın? Segment, adres alanının belirli bir uzunluktaki bitişik kısmıdır ve standart adres alanımızda mantıksal olarak farklı üç segmentimiz vardır: **code**, **stack** ve **heap**.



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

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. Şekil 16.1'deki adres alanını fiziksel belleğe yerleştirmek istediğimizi varsayalım. Segment başına bir taban ve sınır çifti ile, her bir segmenti bağımsız olarak fiziksel belleğe yerleştirebiliriz. Örneğin, bkz. Şekil 16.2 (sayfa 3); orada, içinde bu üç segment bulunan (ve işletim sistemi için ayrılmış 16 KB) 64 KB'lik bir fiziksel bellek görüyorsunuz.



Şekil 16.2: **Segmentleri Fiziksel Belleğe Yerleştirme**

Diyagramda görebileceğiniz gibi, fiziksel bellekte yalnızca kullanılan bellek alanı ayrılır ve böylece büyük miktarda kullanılmayan adres alanı (bazen seyrek adres alanları(**sparse address spaces**) olarak adlandırdığımız) ile büyük adres alanları barındırılabilir.

Bu şekilden görülebileceği gibi, kod segmenti 32KB fiziksel adresinde bulunur ve 2KB büyüklüğündedir, heap segmenti ise 34KB'de bulunur ve 3KB büyüklüğündedir. Bu durumda, segment boyutu önceki olarak tanıtılan bounds register ile aynıdır; donanıma bu segmentin içinde geçerli olan bayt sayısını tam olarak bildirir ve böylece donanımın bir programın bu sınırların dışına geçersiz bir erişim yaptığını tespit etmesine olanak sağlar.

Segment	Base	Size
Code	32K	2K
Heap	34K	3K
Stack	28K	2K

Şekil 16.3: **Segment Kayıt Değerleri**

Kod segmentinin 32 KB fiziksel adrese yerleştirildiğini ve 2 KB boyutunda olduğunu ve heap segmentinin 34 KB olarak yerleştirildiğini ve 3 KB olduğunu şekilden 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 sınırlar dışında yasadışı bir erişim yaptığında bunu belirlemesini sağlar).

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

SEGMENTASYON HATASI

Bölünmüş hata(**segmentation fault**) veya ihlal terimi, bölümlenmiş bir makinede geçersiz bir adrese erişen bir bellek erişiminden kaynaklanmaktadır. Komedi unsuru olarak, bu terim bölümlemeyi destekleyen hiçbir makinede bile devam etmektedir. Ya da bu kadar komik değilse, eğer kodunuzun neden sürekli hatalı olduğunu anlayamıyorsanız.

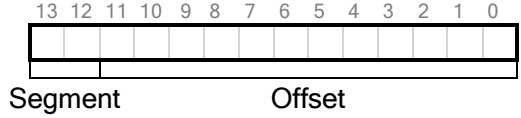
bir referans yapıldığını varsayalım. Referans gerçekleştiğinde (diyelim ki bir komut getirmede), donanım, istenen fiziksel adrese ulaşmak için bu segmentteki (bu durumda 100) ofsete temel değeri ekleyecektir: 100 + 32KB veya 32868. sonra adresin sınırlar içinde olduğunu kontrol edin (100, 2 KB'den küçüktür), öyle olduğunu bulun ve fiziksel bellek adresi 32868'e referans verin.

Şimdi heapteki bir adrese bakalım, sanal adres 4200 (yine Şekil 16.1'e bakın). Yığın tabanına (34KB) sanal adres 4200'ü eklersek, doğru fiziksel adres olmayan 39016 fiziksel adresini elde ederiz. İlk yapmamız gereken şey, ofseti heape çıkarmak, yani adresin bu segmentte hangi bayt(lar)a atıfta bulunduğunu. Heap sanal adres 4KB'de (4096) başladığından, 4200'ün 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.

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

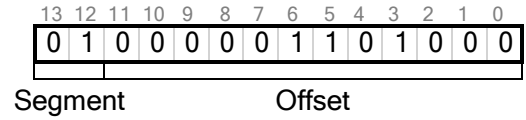
16.2 Hangi Segmente Referans Yapıyoruz?

Donanım, çeviri sırasında segment kayıtlarını kullanır. Bir segmentin ofsetini ve bir adresin hangi segmente atıfta bulunduğunu nasıl bilir? Bazen açık(**explicit**) bir yaklaşım olarak adlandırılan ortak bir yaklaşım, sanal adresin en üstteki birkaç bitine dayalı olarak adres alanını bölümlere ayırmaktır; bu teknik VAX/VMS sisteminde [LL82] kullanıldı. Yukarıdaki örneğimizde üç segmentimiz var; bu nedenle görevimizi gerçekleştirmek için iki bite ihtiyacımız var. Segmenti seçmek için 14 bitlik sanal adresimizin ilk iki bitini kullanırsak, sanal adresimiz şöyle görünür:



Örneğimizde, en üstteki iki bit 00 ise, donanım sanal adresin kod bölümünde olduğunu bilir ve bu nedenle, adresi doğru fiziksel konuma yeniden yerleştirmek için kod tabanı ve sınır çiftini kullanır. En üstteki iki bit 01 ise, donanım adresin heapte olduğunu bilir ve böylece heap tabanını ve sınırlarını kullanır. Bunun net olduğundan emin olmak için yukardan(4200) örnek heap sanal adresimizi alıp

çevirelim. İkili biçimde sanal adres 4200 burada görülebilir:



Resimden de görebileceğiniz gibi üstteki iki bit (01) donanıma hangi segmentten bahsettiğimizi söylüyor. Alttaki 12 bit, segmentin ofsetidir: 0000 0110 1000 veya hex 0x068 veya ondalık olarak 104. Böylece donanım, hangi segment kaydının kullanılacağını belirlemek için ilk iki biti alır ve ardından sonraki 12 biti segmentin ofseti olarak alır. Taban kaydını ofsete ekleyerek, donanım nihai fiziksel adrese ulaşır. Ötelemenin sınır denetimini de kolaylaştırdığına dikkat edin: ötelemenin sınırlardan küçük olup olmadığını kolayca kontrol edebiliriz; değilse, adres geçersizdir. 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. Spesifik olarak SEG_MASK, 0x3000'e, SEG_SHIFT'e 12'ye ve OFSET_MASK'e 0xFFF'ye ayarlanacaktır.

Ayrıca, en üstteki iki biti kullandığımızda ve yalnızca üç parçamız (code, heap, stack) olduğunda, adres alanının bir bölümünün kullanılmadığını fark etmiş olabilirsiniz. Sanal adres alanını tam olarak kullanmak (ve kullanılmayan bir segmentten kaçınmak) için, bazı sistemler kodu heaple aynı segmente koyar ve böylece hangi segmentin kullanılacağını [LL82] seçmek için yalnızca bir bit kullanır.

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

Donanımın belirli bir adresin hangi segmentte olduğunu belirlemesinin başka yolları da vardır. Örtük(**implicit**) yaklaşımda donanım, adresin nasıl oluştuğunu fark ederek segmenti belirler. Örneğin, adres program sayacından oluşturulmuşsa (yani bu bir komut getirme işlemiyse), o zaman adres kod segmenti içindedir; adres stack veya temel işaretçiyi(base pointer) temel alıyorsa, stack segmentinde olmalıdır; diğer tüm adresler heapte olmalıdır.

16.3 Peki Ya Stack?

Şimdiye kadar, adres uzayının önemli bir bileşenini dışarıda bıraktık: stack. Stack, yukarıdaki diyagramda 28 KB fiziksel adresine taşınmıştır, ancak kritik bir farkla: geriye doğru büyür (yani, daha düşük adreslere doğru). Fiziksel bellekte, 28KB'de "başlar" ve 16 KB ila 14 KB sanal adreslere karşılık gelen 26 KB'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 yönde 0'a ayarlanan bir bit). Şekil 16.4'te donanımın izlediklerine ilişkin güncellenmiş görünümümüz:

Segment	Base	Size (max 4K)	Grows Positive?
$Code_{00}$	32K	2K	1
$Heap_{01}$	34K	3K	1
$Stack_{11}$	28K	2K	0

Şekil 16.4: **Segment Kayıtları (Negatif Büyüme Destekli)**

Segmentlerin negatif yönde büyüyebileceğine dair donanım anlayışıyla, donanımın artık bu tür sanal adresleri biraz farklı çevirmesi gerekir. İşlemi anlamak için örnek bir stack sanal adresi alıp çevirelim. Bu örnekte, fiziksel adres 27KB ile eşleşmesi gereken sanal adres 15KB'ye erişmek istediğimizi varsayalım. İkili biçimdeki sanal adresimiz şuna benzer: 11 1100 0000 0000 (onaltılık 0x3C00). Donanım, segmenti belirlemek için en üstteki iki biti (11) kullanır, ancak sonra 3 KB'lik bir ofsetle baş başa kalırız. Doğru negatif uzaklığı elde etmek için, maksimum segment boyutunu 3KB'den çıkarmalıyız: bu örnekte, bir segment 4KB olabilir ve dolayısıyla doğru negatif ofset 3KB eksi 4KB'dir, bu da -1KB'ye eşittir. Doğru fiziksel adrese ulaşmak için tabana (28KB) negatif uzaklığı (-1KB) ekliyoruz: 27KB. Sınır kontrolü, negatif ofsetin mutlak değerinin segmentin geçerli boyutundan (bu durumda 2 KB) küçük veya ona eşit olması sağlanarak hesaplanabilir.

[1]Stack basit olsun diye 28 KB'de "başlıyor" desek de, bu değer aslında geriye doğru büyüyen bölgenin konumunun hemen altındaki bayttır; ilk geçerli bayt aslında 28KB eksi 1'dir. Bunun tersine, ileriye doğru büyüyen bölgeler, segmentin ilk baytının adresinde 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 tasarımcıları biraz daha fazla donanım desteğiyle yeni verimlilik türlerini gerçekleştirebileceklerini kısa sürede fark ettiler. Spesifik olarak, bellekten tasarruf etmek için bazen belirli bellek bölümlerini adres alanları arasında paylaşmak yararlı olabilir. Özellikle, kod paylaşımı(**code sharing**) yaygındır ve günümüzde sistemlerde hala kullanılmaktadır.

Paylaşımı desteklemek için, donanımdan koruma bitleri(**protection bits**) 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 gösterir. Bir kod segmentini salt okunur olarak ayarlayarak, aynı kod, izolasyona zarar verme endişesi olmadan birden fazla işlem arasında paylaşılabilir; her process hala kendi özel hafızasına eriştiğini düşünürken, işletim sistemi process tarafından değiştirilemeyen hafızayı gizlice paylaşıyor ve böylece yanılsama korunmuş oluyor.

Donanım (ve işletim sistemi) tarafından izlenen ek bilgilerin bir örneği Şekil 16.5'te gösterilmektedir. Gördüğünüz gibi, kod bölümü okumaya ve yürütmeye ayarlanmıştır ve böylece bellekteki aynı fiziksel bölüm, birden çok sanal adres alanına eşlenebilir.

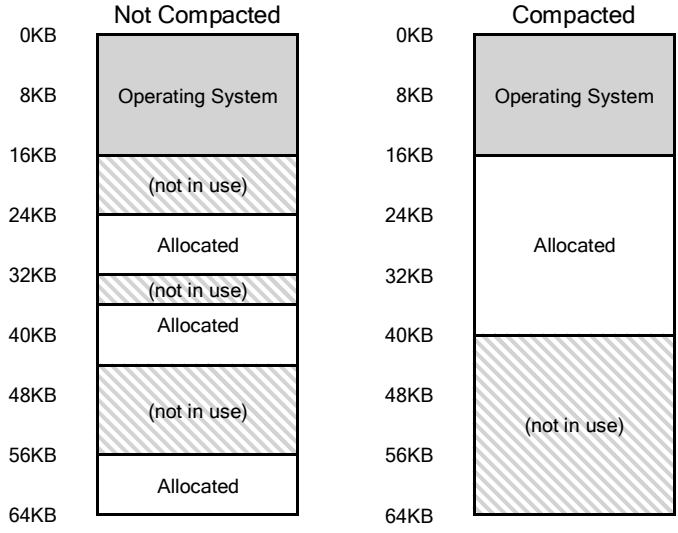
Segment	Base	Size (max 4K)	Grows Positive?	Protection
$Code_{00}$	32K	2K	1	Read-Execute
$Heap_{01}$	34K	3K	1	Read-Write
$Stack_{11}$	28K	2K	0	Read-Write

Şekil 16.5: **Segment Kayıt Değerleri (Korumalı)**

Koruma bitleriyle, daha önce açıklanan donanım algoritmasının da değişmesi gerekir. Donanım, bir sanal adresin sınırlar içinde olup olmadığını kontrol etmenin yanı sıra, belirli bir erişime izin verilip verilmediğini de kontrol etmelidir. Bir kullanıcı işlemi salt okunur bir kesime yazmaya veya çalıştırılamayan bir kesimden yürütmeye çalışırsa, donanım bir istisna oluşturmalı ve böylece işletim sisteminin rahatsız edici işlemle ilgilenmesine izin vermelidir.

16.5 İnce-taneli vs. Kalın-taneli Segmentasyon

Şimdiye kadar verdiğimiz örneklerin çoğu, yalnızca birkaç segmentli (kod, yığın, yığın) sistemlere odaklandı; adres alanını nispeten büyük, kaba parçalara böldüğü için bu bölümlemeyi kaba taneli(**coarse-grained**) olarak düşünebiliriz. Bununla birlikte, bazı eski sistemler (örneğin, Multics [CV65, DD68]) daha esnekti ve ince taneli(**fine-grained**) bölümleme olarak adlandırılan çok sayıda küçük bölümden oluşan adres alanlarına izin verildi. Birçok segmenti desteklemek, bellekte saklanan bir tür segment tablosu ile daha da fazla donanım desteği gerektirir. Bu tür bölüm tabloları çok sayıda parçanın oluşturulmasını destekler ve böylece bir sistemin bölümleri şimdiye kadar tartıştığımızdan daha esnek şekillerde kullanmasını sağlar. Örn, Burroughs B5000 gibi eski makineler binlerce segmenti destekliyordu ve bir derleyicinin kesme işlemini gerçekleştirmesini bekliyordu.



Şekil 16.6: **Sıkıştırılmamış ve Sıkıştırılmış Bellek**

kodu ve verileri işletim sistemi ve donanımın [RK68] destekleyeceği ayrı bölümlere ayırın. O zamanki düşünce, ince taneli bölümlere sahip olarak, işletim sisteminin hangi bölümlerin kullanımda olduğunu ve hangilerinin kullanılmadığını daha iyi öğrenebileceği ve böylece ana belleği daha verimli kullanabileceğiydi.

16.6 İşletim Sistemi(OS) Desteği

Artık segmentasyonun nasıl çalıştığına dair temel bir fikre sahip olmalısınız. Adres alanının parçaları, sistem çalışırken fiziksel belleğe yeniden yerleştirilir ve bu nedenle, tüm adres alanı için yalnızca tek bir taban/sınır çifti içeren daha basit yaklaşımımıza göre çok büyük bir fiziksel bellek tasarrufu elde edilir. Spesifik olarak, stack ile heap arasındaki tüm kullanılmayan alanın fiziksel belleğe ayrılması gerekmez, bu da fiziksel belleğe daha fazla adres alanı sığdırmamıza ve işlem başına büyük ve seyrek bir sanal adres alanını desteklememize olanak tanır.

Ancak segmentasyon, işletim sistemi için bir dizi yeni sorunu gündeme getirir. İlki eskidir: İşletim sistemi bir bağlam anahtarında ne yapmalıdır? 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 küçüldüğünde) işletim sistemi etkileşimidir. Örneğin, bir program bir nesneyi tahsis etmek için malloc()'u çağırabilir. Bazı durumlarda, varolan heap isteğe hizmet verebilir ve böylece

İPUCU: 1000 ÇÖZÜM VARSA, BÜYÜK ÇÖZÜM OLMAZ
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 parçalar halinde ayırmayarak sorunu tamamen ortadan kaldırmaktır.

malloc(), nesne için boş alan bulur ve arayana bir işaretçi döndürür. Ancak diğerlerinde heap segmentinin kendisinin büyümesi gerekebilir. Bu durumda, bellek ayırma kitaplığı heapi 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 kitaplığı başarı hakkında bilgilendirir; kitaplık daha sonra yeni nesne için yer ayırabilir ve başarılı bir şekilde çağıran programa geri dönebilir. Kullanılabilir fiziksel bellek yoksa veya arama işleminin zaten çok fazla belleğe sahip olduğuna karar verirse işletim sisteminin isteği reddedebileceğini unutmayın.

Son ve belki de en önemli konu, fiziksel bellekte boş alanı yönetmektir. Yeni bir adres alanı oluşturulduğunda, işletim sistemi, bölümleri için fiziksel bellekte boş bir alan bulabilmelidir. Önceki olarak, her adres alanının aynı boyutta olduğunu varsaydık ve böylece fiziksel bellek işlemleri sığdıracak bir takım yuvalar olarak düşünülebilirdi. Şimdi, her işlem için birkaç bölümümüz var ve her bölüm farklı bir boyutta olabilir.

Ortaya çıkan genel sorun, fiziksel belleğin kısa sürede küçük boş alan boşluklarıyla dolması, yeni bölümler ayırmayı veya var olanları büyütmeyi zorlaştırmasıdır. Bu soruna dış parçalanma(**external fragmentation**) [R69] diyoruz; bkz. Şekil 16.6 (sol).

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

Bu soruna bir çözüm, mevcut bölümleri yeniden düzenleyerek fiziksel belleği sıkıştırmak olacaktır. Örneğin, işletim sistemi hangi işlemlerin çalıştığını durdurabilir, verilerini bitişik bir bellek bölgesine kopyalayabilir, segment kayıt değerlerini yeni fiziksel konumları işaret edecek şekilde değiştirebilir ve böylece çalışmak için geniş bir boş bellek alanına sahip olabilir. . Bunu yaparak, işletim sistemi yeni tahsis talebinin başarılı olmasını sağlar. Bununla birlikte, bölümleri kopyalamak yoğun bellek gerektirdiğinden ve genellikle makul miktarda işlemci süresi kullandığından sıkıştırma pahalıdır; Sıkıştırılmış fiziksel belleğin bir diyagramı için bkz. Şekil 16.6 (sağ). Sıkıştırma aynı zamanda (ironik bir şekilde) mevcut segmentleri büyütme isteklerini hizmet vermeyi zorlaştırır ve bu nedenle bu tür istekleri 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** (boş alanların bir listesini tutar ve istek sahibine istenen tahsisi sağlayan boyut olarak en yakın olanı döndürür), **worst-fit**, **first-fit** gibi klasik algoritmalar da dahil olmak üzere, insanların benimsediği kelimenin tam anlamıyla yüzlerce yaklaşım vardır. ve **buddy algoritması** [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, harici parçalanma yine de var olacaktır; bu nedenle, iyi bir algoritma basitçe onu en aza indirmeye çalışır.

16.7 Özet

Bölümleme, bir dizi sorunu çözer ve belleğin daha etkili bir şekilde sanallaştırılmasını oluşturmamıza yardımcı olur. Yalnızca dinamik yer değiştirmenin ötesinde, bölümleme, adres alanının mantıksal bölümleri arasındaki büyük potansiyel bellek israfını önleyerek seyrek adres alanlarını daha iyi destekleyebilir. Aritmetik segmentasyonun gerektirdiği kolay ve donanıma çok uygun olduğundan, aynı zamanda hızlıdır; çeviri masrafları minimum düzeydedir. Bir yan fayda da ortaya çıkıyor: kod paylaşımı. Kod ayrı bir segmente yerleştirilirse, böyle bir segment çalışan birden çok program arasında potansiyel olarak paylaşılabilir.

Ancak öğrendiğimiz gibi, değişken boyutlu segmentleri bellekte tahsis etmek, üstesinden gelmek istediğimiz bazı sorunlara yol açıyor. Birincisi, yukarıda tartışıldığı gibi, dış parçalanmadır. Segmentler değişken boyutlu olduğundan, boş bellek tek boyutlu parçalara bölünür ve bu nedenle bir bellek ayırma talebini karşılamak zor olabilir. Akıllı algoritmalar [W+95] veya periyodik olarak sıkıştırılmış bellek kullanılmaya çalışılabilir, ancak sorun temeldir ve kaçınılması zordur.

İkinci ve belki de daha önemli sorun, segmentasyonun hala tamamen genelleştirilmiş, seyrek adres alanımızı destekleyecek kadar esnek olmamasıdır. Örneğin, tümü tek bir mantıksal segmentte bulunan büyük ama seyrek kullanılan bir yığınımız varsa, erişilebilmesi için tüm yığının yine de bellekte bulunması gerekir. Başka bir deyişle, adres alanının nasıl kullanıldığına ilişkin modelimiz, temeldeki bölümlemenin onu desteklemek için nasıl tasarlandığına tam olarak uymuyorsa, bölümleme pek iyi çalışmaz. Bu nedenle bazı yeni çözümler bulmamız gerekiyor. Onları bulmaya hazır mısın?

Referanslar

[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. Communications 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. Available: 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. Communications 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. Communications of the ACM, Volume 12:7, July 1969. *One of the earliest papers to discuss fragmentation.*

[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, segmentasyonla bir sistemde adres çevirilerinin nasıl gerçekleştirildiğini görmek için kullanılabilir. Ayrıntılar için README dosyasına bakın.

Sorular

1. İlk olarak küçük bir adres alanı kullanarak bazı adresleri çevireceğiz.

   Burada birkaç farklı rastgele tohumla(seed) birlikte basit bir parametre kümesi bulunmaktadır; 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
```

   Bu kod parçacığı, 128 byte adres alanına sahip bir bilgisayar simülasyonu oluşturur. Bu simülasyon, 512 byte fiziksel bellek boyutuna sahiptir ve iki ayrı bölümden oluşur. İlk bölümün temel kaydırıcısı 0 ve sınır kaydırıcısı 20, ikinci bölümün temel kaydırıcısı 512 ve sınır kaydırıcısı 20'dir.

   Kullanıcı, programı çalıştırmak için belirtilen rastgele seed değerini kullanarak programı çalıştırabilir. Program, belirtilen adresleri çevirir ve kullanıcıya geçerli olup olmadıklarını ve hangi bölümde bulunduklarını gösterir.

```
root@light-ubuntu:~/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?
```

```
root@light-ubuntu:~/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?
```

Ekran görüntüsünde de görüldüğü üzere '-s' yani seed parametresi, kullanıcıdan aldığı integer değere göre random kütüphanesinden random.seed() fonksiyonunu kullanarak bir seed değeri belirler. Daha sonra bunu sanal adresin yerine belirtmek için kullanır.

Kullanıcı her farklı seed parametresi girdiğinde ekran görüntüsünde de görüldüğü üzere farklı sanal adres alanları çıkar.

2. Şimdi, oluşturduğumuz bu küçük adres alanını (yukarıdaki sorudaki parametreleri kullanarak) anlayıp anlamadığımıza bakalım. Segment 0'daki en yüksek yasal sanal adres nedir? Peki ya segment 1'deki en düşük yasal sanal adres? Tüm bu adres uzayındaki en düşük ve en yüksek yasadışı adresler nelerdir? Son olarak, haklı olup olmadığınızı test etmek için segmentation.py'yi -A bayrağıyla nasıl çalıştırırsınız?

   İlk bölümün en yüksek geçerli sanal adresi 20'dir. İkinci bölümün en düşük geçerli sanal adresi 512'dir. Bu adres alanının en düşük ve en yüksek geçersiz adresleri 0 ve 532'dir. Kullanıcı, aşağıdaki komutu (segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -A "0, 20, 512, 532") kullanarak adresleri test etmek için programı çalıştırabilir:

```
root@light-ubuntu:~/ostep-homework/vm-segmentation# python3 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -A "0, 20, 512, 532"
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: 0x00000000 (decimal:    0) ⟶ PA or segmentation violation?
  VA  1: 0x00000014 (decimal:   20) ⟶ PA or segmentation violation?
```

3. 128 bayt fiziksel bellekte 16 baytlık küçücük bir adres alanımız olduğunu varsayalım. Simülatörün belirtilen adres akışı için aşağıdaki çeviri sonuçlarını oluşturmasını sağlamak için hangi taban ve sınırları ayarlarsınız: valid, valid, violation, ..., violation, valid, valid? Aşağıdaki parametreleri varsayın:

```
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 ?
```

Ekran görüntüsündeki komut, bir adres alanında 16 baytlık bir alan için bir fiziksel bellek boyutunu 128 baytlık bir alana ayarlar.

Kullanıcı, aşağıdaki parametreleri kullanarak istenen çeviri sonuçlarını elde etmek için taban ve sınırları ayarlayabilir:

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 8 --b1 8 --l1 8

Bu konfigürasyon, 0-7 arasındaki adresler için geçerli, 8-15 arasındaki adresler için geçersiz ve 16-23 arasındaki adresler için geçerli olacaktır.

```
root@light-ubuntu:~/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 8 --b1 8 --l1 8
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                  : 8

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

Virtual Address Trace
  VA  0: 0x00000000 (decimal:    0) ⟶ PA or segmentation violation?
  VA  1: 0x00000001 (decimal:    1) ⟶ PA or segmentation violation?
  VA  2: 0x00000002 (decimal:    2) ⟶ PA or segmentation violation?
  VA  3: 0x00000003 (decimal:    3) ⟶ PA or segmentation violation?
  VA  4: 0x00000004 (decimal:    4) ⟶ PA or segmentation violation?
  VA  5: 0x00000005 (decimal:    5) ⟶ PA or segmentation violation?
  VA  6: 0x00000006 (decimal:    6) ⟶ PA or segmentation violation?
  VA  7: 0x00000007 (decimal:    7) ⟶ PA or segmentation violation?
  VA  8: 0x00000008 (decimal:    8) ⟶ PA or segmentation violation?
  VA  9: 0x00000009 (decimal:    9) ⟶ PA or segmentation violation?
  VA 10: 0x0000000a (decimal:   10) ⟶ PA or segmentation violation?
  VA 11: 0x0000000b (decimal:   11) ⟶ PA or segmentation violation?
  VA 12: 0x0000000c (decimal:   12) ⟶ PA or segmentation violation?
  VA 13: 0x0000000d (decimal:   13) ⟶ PA or segmentation violation?
  VA 14: 0x0000000e (decimal:   14) ⟶ PA or segmentation violation?
  VA 15: 0x0000000f (decimal:   15) ⟶ 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 (topbit0) or segment 1 (topbit1). 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.
```

4. Rastgele oluşturulmuş sanal adreslerin kabaca %90'ının geçerli olduğu (segmentasyon ihlalleri değil) bir sorun oluşturmak istediğimizi varsayalım. Simülatörü bunu yapacak şekilde nasıl yapılandırmalısınız? Bu sonuca ulaşmak için hangi parametreler önemlidir?

   Rastgele oluşturulmuş sanal adreslerin kabaca %90'ının geçerli olduğu bir sorun oluşturmak için, segment kayıtları için taban ve sınırları, sanal adreslerin çoğu sınırlar içinde kalacak şekilde ayarlamalıyız.

   Örneğin, tabanı ve sınırları, 0 kesimi adres alanının çoğunu ve 1. bölümü adres alanının küçük bir bölümünü kaplayacak şekilde ayarlayabiliriz.

   Segment kayıtları için taban ve sınırları ayarlamak için simülatörü çalıştırırken; -b0, -l0, -b1 ve -l1 bayrakları kullanılabilir.

   Örneğin, segment 0'ın tabanını 0'a ve segment 0'ın sınırını 80'e ayarlamak için aşağıdaki komutu kullanabiliriz:

   segmentation.py -b0 0 -l0 80

```
root@light-ubuntu:~/ostep-homework/vm-segmentation# python3 segmentation.py -b0 0 -l0 80
ARG seed 0
ARG address space size 1k
ARG phys mem size 16k

Segment register information:

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

  Segment 1 base  (grows negative) : 0x0000325a (decimal 12890)
  Segment 1 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?

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.
```

Ardından, segment 1 için taban ve sınırları benzer şekilde ayarlayabiliriz.

Örneğin segment 1'in tabanını 80'e ve segment 1'in limitini 20'ye ayarlamak için aşağıdaki komutu kullanabiliriz:

segmentation.py -b1 80 -l1 20

```
root@light-ubuntu:~/ostep-homework/vm-segmentation# python3 segmentation.py -b1 80 -l1 20
ARG seed 0
ARG address space size 1k
ARG phys mem size 16k

Segment register information:

  Segment 0 base  (grows positive) : 0x00000001 (decimal 1)
  Segment 0 limit                  : 1

  Segment 1 base  (grows negative) : 0x0000325a (decimal 12890)
  Segment 1 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?

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.
```

Ayrıca sırasıyla -a ve -p bayraklarını kullanarak adres alanının ve fiziksel belleğin boyutunu da ayarlayabiliriz.

Yapılandırmanızın doğru olup olmadığını test etmek için, çevrilecek adreslerin listesini belirtmek için -A bayrağını kullanabiliriz.

Adreslerin çoğu başarılı bir şekilde çevrildiyse, yapılandırmanız muhtemelen doğrudur.

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

Simülatörü hiçbir sanal adres olmayacak şekilde çalıştırmak için, -l ve -L seçeneklerini kullanarak her iki segmentin uzunluğunu 0'a ayarlayabiliriz.

Bu, herhangi bir segmentin sınırları içinde hiçbir sanal adresin olmadığı ve bu nedenle tüm sanal adreslerin geçersiz olduğu anlamına gelir.

Örneğin, aşağıdaki komut, simülatörü 128 baytlık bir adres alanı ve 512 baytlık bir fiziksel bellekle çalıştırır ve her iki segmentin de uzunluğu 0'dır:

segmentation.py -a 128 -p 512 -l 0 -L 0

```
root@light-ubuntu:~/ostep-homework/vm-segmentation# python3 segmentation.py -a 128 -p 512 -l 0 -L 0
ARG seed 0
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x000001b0 (decimal 432)
  Segment 0 limit                  : 0

  Segment 1 base  (grows negative) : 0x00000184 (decimal 388)
  Segment 1 limit                  : 0

Virtual Address Trace
  VA  0: 0x00000035 (decimal:   53) ⟶ PA or segmentation violation?
  VA  1: 0x00000021 (decimal:   33) ⟶ PA or segmentation violation?
  VA  2: 0x00000041 (decimal:   65) ⟶ PA or segmentation violation?
  VA  3: 0x00000033 (decimal:   51) ⟶ PA or segmentation violation?
  VA  4: 0x00000064 (decimal:  100) ⟶ 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.
```

Alternatif olarak, her iki parçanın tabanını adres alanının dışında olacak şekilde ayarlayabiliriz.

Örneğin, adres alanı 128 bayt ise, her iki segmentin tabanını 128 veya daha yükseğe ayarlamak, herhangi bir segmentin sınırları içinde hiçbir sanal adresin olmamasını sağlar.

segmentation.py -a 128 -p 512 -b 128 -B 128

```
root@light-ubuntu:~/ostep-homework/vm-segmentation# python3 segmentation.py -a 128 -p 512 -b 128 -B 128
ARG seed 0
ARG address space size 128
ARG phys mem size 512

Segment register information:

  Segment 0 base  (grows positive) : 0x00000080 (decimal 128)
  Segment 0 limit                  : 59

  Segment 1 base  (grows negative) : 0x00000080 (decimal 128)
  Segment 1 limit                  : 56

Virtual Address Trace
  VA  0: 0x00000035 (decimal:   53) ⟶ PA or segmentation violation?
  VA  1: 0x00000021 (decimal:   33) ⟶ PA or segmentation violation?
  VA  2: 0x00000041 (decimal:   65) ⟶ PA or segmentation violation?
  VA  3: 0x00000033 (decimal:   51) ⟶ PA or segmentation violation?
  VA  4: 0x00000064 (decimal:  100) ⟶ PA or segmentation violation?
```