16

Segmentasyon

Şimdiye kadar her işlemin tüm adres alanını belleğe koyuyorduk. Temel ve sınır kayıtları ile işletim sistemi, işlemleri fiziksel belleğin farklı bölümlerine kolayca yeniden yerleştirebilir. Ancak, bu adres alanlarımızda ilginç bir şey fark etmiş olabilirsiniz tam ortada, yığın (stack) ile küme (heap) arasında büyük bir "boş" alan parçası var.

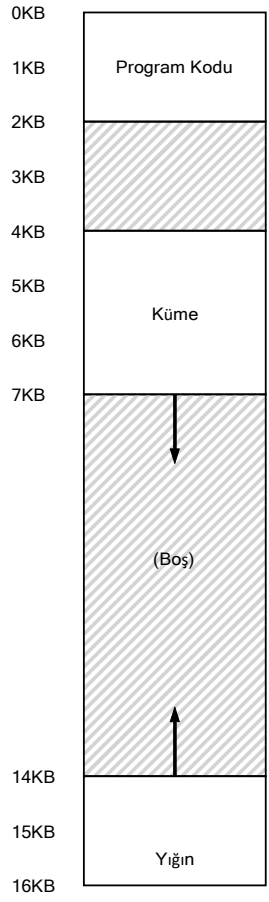
Şekil 16.1'den tahmin edebileceğiniz gibi, yığın ve yığın arasındaki boşluk işlem tarafından kullanılmasa da, tüm adres alanını fiziksel bellekte bir yere yeniden yerleştirdiğimizde, yine de fiziksel belleği kaplıyor.; bu nedenle, belleği sanallaştırmak için bir taban ve sınırlar kayıt çifti kullanmanın basit yaklaşımı israftır. Ayrıca, tüm adres alanı belleğe sığmadığında bir programı çalıştırmayı oldukça zorlaştırır; bu nedenle taban ve sınırlar istediğimiz kadar esnek değil. Ve böylece:

KRİTİK KONU GENİŞ BİR ADRES ALANI NASIL DESTEKLENİR

Yığın ve küme 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 Taban/Sınırlar

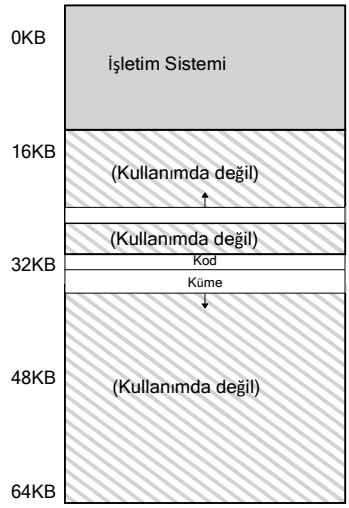
Bu sorunu çözmek için bir fikir doğdu ve buna **segmentasyon (segmentation)** deniyor. Bu, en azından 1960'ların [H61, G62] başlarına kadar uzanan oldukça eski bir fikirdir. Fikir basit: MMU'muzda yalnızca bir taban ve sınır çiftine sahip olmak yerine, neden adres uzayının mantıksal segmenti (bölüm)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



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

adres alanında, mantıksal olarak farklı üç segmentimiz var: kod, yığın ve küme. 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 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.

Segmentasyonu desteklemek için MMU'muzdaki donanım yapısı tam olarak beklediğiniz gibidir: bu durumda, üç temel ve sınır kayıt çifti seti. Aşağıdaki Şekil 16.3, yukarıdaki örnek için kayıt değerlerini göstermektedir; her sınır kaydı, bir segmentin boyutunu tutar.

Segment	Temel	Boyut
Kod	32K	2K
Küme	34K	3K
Yığın	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) bir referans yapıldığını varsayalım. Referans gerçekleştiğinde

AYRICA: SEGMENTASYON HATASI

Segmentasyon hatası (segmentation fault) veya ihlali terimi, segmentlere ayrılmış bir makinede geçersiz bir adrese yapılan bellek erişiminden kaynaklanır. Esprili bir şekilde, terim, segmentasyon desteği olmayan makinelerde bile devam ediyor. Veya kodunuzun neden sürekli hatalı olduğunu anlayamıyorsanız, o kadar komik değil.

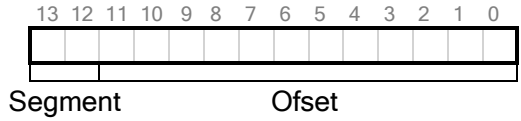
(diyelim ki bir talimat getirmede), donanım istenen fiziksel adrese ulaşmak için bu segmentteki (bu durumda 100) ofsete temel değeri ekleyecektir: 100 + 32KB veya 32868. Daha sonra adresin sınırlar içinde olup olmadığını kontrol edecek (100, 2 KB'den küçüktür), olduğunu bulacak ve fiziksel bellek adresi 32868'e referans verecektir.

Şimdi yığındaki 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 yığına çıkarmak, yani adresin bu segmentte hangi bayt(lar)a atıfta bulunduğunu. Yığın 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.

Yığının 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 (segmentationfault)**.

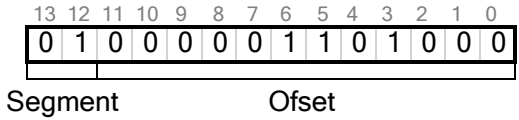
16.2 Hangi Segmentten Bahsediyoruz?

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 yaygın 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 dresin kod bölümünde olduğunu bilir ve bu nedenle, adresi doğru iziksel konuma yeniden yerleştirmek için kod tabanı ve sınır çiftini ullanır. En üstteki iki bit 01 ise, donanım adresin yığında olduğunu ilir,

ve böylece yığın tabanını ve sınırlarını kullanır. Bunun net olduğundan emin olmak için yukarıdan (4200) örnek yığın sanal adresimizi alıp çevirelim. İkili biçimde sanal adres 4200 burada görülebilir:



Resimden de görebileceğiniz gibi üstteki iki bit (01) donanıma hangi segmentten bahsettiğimizi söyler. 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. Temel kaydı 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. Özellikle, `SEG MASK` ayarlanacak `0x3000`, `SEG SHIFT` ile `12`, ve `OFFSET MASK` to `0xFFF`.

Ayrıca, en üstteki iki biti kullandığımızda ve yalnızca üç parçamız (kod, küme, 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 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 4KB olan bir maksimum boyutla sınırlıdır (segmentleri seçmek için en üstteki iki bitin kullanılması, 16KB adres alanının dört parçaya veya bu örnekte 4KB'ye bölündüğü anlamına gelir). Çalışan bir program, bir segmenti (yığın veya yığın 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 oluştuğuna dikkat ederek segmenti belirlerÖrneğin, adres program sayacından oluşturulmuşsa (yani bu bir komut getirme işlemiyse), o zaman adres kod segmenti içindedir; adres yığın veya temel işaretçiyi temel alıyorsa, yığın segmentinde olmalıdır; diğer tüm adresler kümede olmalıdır.

16.3 Yığın Ne Olacak?

Şimdiye kadar, adres uzayının önemli bir bileşenini dışarıda bıraktık: yığın. Yığın, yukarıdaki şemada 28 KB fiziksel adresine taşınmıştır, ancak kritik bir farkla: geriye doğru büyür (yani, daha düşük adreslere doğru). Fiziksel bellekte, 28KB1'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	Temel	Boyut (maks 4K)	Pozitif Büyür Mü?
Kod_{00}	32K	2K	1
$Küme_{01}$	34K	3K	1
$Yığın_{11}$	28K	2K	0

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

Segmentlerin negatif yönde büyüyebileceğine dair donanım anlayışıyla, donanım artık bu tür sanal adresleri biraz farklı çevirmek zorundadır. İşlemi anlamak için örnek bir yığın 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 bu nedenle 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]Basit olması için yığının 28 KB'de "başladığını" söylesek 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 uzaklıktır.

16.4 Paylaşım Desteği

Segmentasyon desteği arttıkça, sistem tasarımcıları kısa sürede biraz daha fazla donanım desteğiyle yeni verimlilik türlerini gerçekleştirebileceklerini fark ettiler. Spesifik olarak, bellekten tasarruf etmek için bazen belirli bellek bölümlerini adres alanları arasında **paylaşmak (share)** 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 proses hala kendi özel hafızasına eriştiğini düşünürken, işletim sistemi proses 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	Temel	Boyut (maks 4K)	Pozitif Büyür Mü?	Koruma
Kod_{00}	32K	2K	1	Oku-Yürüt
$Küme_{01}$	34K	3K	1	Oku-yaz
$Yığın_{11}$	28K	2K	0	Oku-Yaz

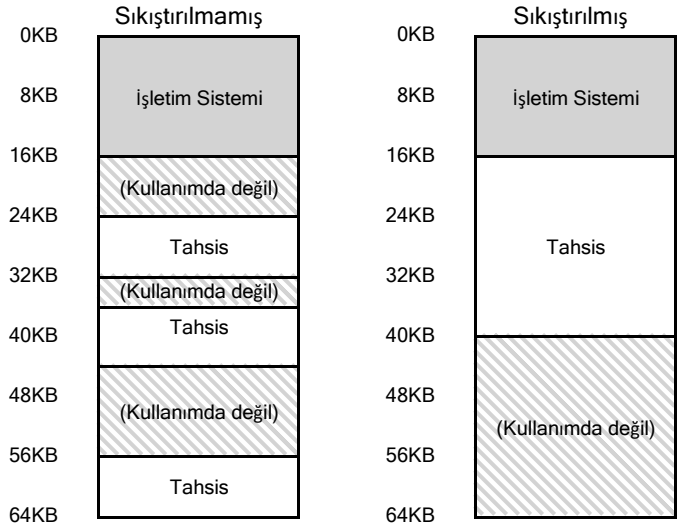
Şekil 16.5: **Segment Kayıt Değerleri (Korumalı)**

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 ve Kaba Taneli Segmentasyon

Şimdiye kadar verdiğimiz örneklerin çoğu, yalnızca birkaç segmentli (yani kod, yığın, Küme) 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 bölümleme olarak adlandırılan çok sayıda küçük bölümden oluşan adres alanlarına izin verildi.

Birçok segmenti desteklemek, bellekte saklanan bir tür **segment tablosu(segment table)** ile daha da fazla donanım desteği gerektirir.. Bu tür parça tabloları genellikle çok fazla sayıda parçanın oluşturulmasını destekler ve böylece bir sistemin bölümleri şimdiye kadar tartıştığımızdan daha esnek şekillerde kullanmasını sağlar. Örneğin, Burroughs B5000 gibi eski makineler binlerce segmenti destekliyordu ve bir derleyicinin kesme işlemini gerçekleştirmesibekliyordu



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 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, yığın ile yığın 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 belki küçüldüğünde) işletim sistemi etkileşimidir. Örneğin, bir program bir nesneyi tahsis etmek için malloc()'u çağırabilir. Bazı durumlarda varolan yığın 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 yığın segmentinin kendisinin büyümesi gerekebilir. Bu durumda, bellek ayırma kitaplığı yığını büyütmek için bir sistem çağrısı gerçekleştirir (ör. 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 sorun, fiziksel bellekteki boş alanı yönetmektir. Yeni bir adres alanı oluşturulduğunda, işletim sistemi segmentleri için fiziksel bellekte yer bulabilmelidir. Daha önce, her bir adres alanının aynı boyutta olduğunu ve bu nedenle fiziksel belleğin, işlemlerin sığacağı bir grup yuva olarak düşünülebileceğini varsaymıştık. Şimdi, işlem başına bir dizi segmentimiz var ve her segment 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 (compact)** 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 kullanıldığından sıkıştırma pahalıdır; görmek

Sıkıştırılmış fiziksel belleğin bir diyagramı için Şekil 16.6 (sağda). 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. **En uygun(Best-fit)** gibi klasik algoritmalar da dahil olmak üzere (boş alanların bir listesini tutan ve talep sahibine istenen tahsisi karşılayan boyut olarak en yakın olanı döndüren) gibi klasik algoritmalar da dahil olmak üzere, insanların benimsediği kelimenin tam anlamıyla yüzlerce yaklaşım vardır **en kötü uyan(worst-fit), ilk uyan(first-fit)**, ve **arkadaş algoritması**(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, harici parçalanma yine de var olacaktır; bu nedenle, iyi bir algoritma basitçe onu en aza indirmeye çalışır.

16.7 Özet

Segmentasyon, 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 minimumdur. 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, bellekte değişken boyutlu segmentler 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 hala 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 adres çevirilerinin segmentasyonlu bir sistemde nasıl yapıldığını görmemizi sağlar. Ayrıntılar için README'ye bakın.

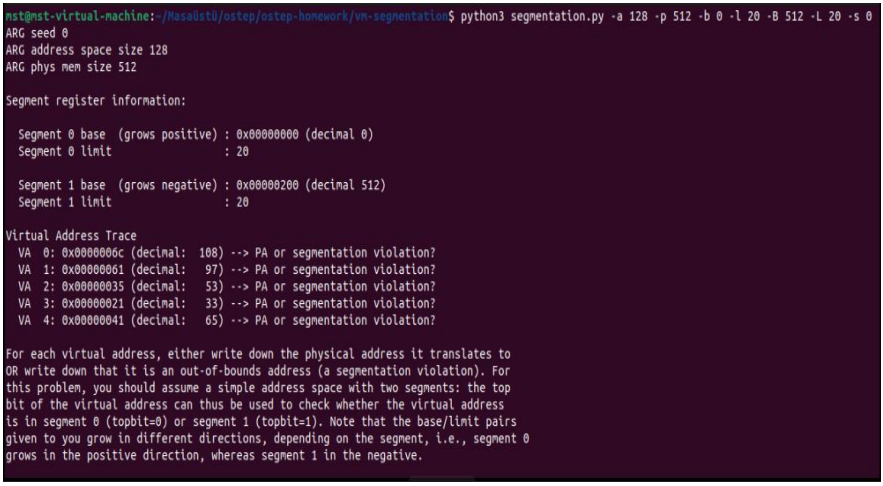
Sorular

1. Önce bazı adresleri çevirmek için küçük bir adres alanı kullanalım. İşte birkaç farklı rasgele tohum içeren basit bir parametre seti; adresleri çevirebilir misin?

```
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
```

Cevap:

Segment = 0 için;



Segment = 1 için;

```
mst@mst-virtual-machine:~/Masaüstü/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.
```

Segment 2 için;

```
mst@mst-virtual-machine:~/Masaüstü/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.
```

2. Şimdi oluşturduğumuz bu küçücük adres uzayını (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?

Cevap:

Segment 0'daki en yüksek yasal sanal adres 0x0000006c'dir.
segment 1'deki en düşük yasal sanal adres 0x00000011'dir.

Tüm adres uzayınaki en yüksek;0x0000007a (segment ikide)
Tüm adres uzayındaki en düşük;0x00000007 (segment ikide)

A0 için;

```
mst@mst-virtual-machine:~/Masaüstü/ostep/ostep-homework/vm-segmentation$ python3 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0 -A 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: 0x00000000 (decimal:    0) --> 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.
```

A1 için;

```
mst@mst-virtual-machine:~/Masaüstü/ostep/ostep-homework/vm-segmentation$ python3 segmentation.py -a 128 -p 512 -b 0 -l 20 -B 512 -L 20 -s 0 -A 1
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: 0x00000001 (decimal:    1) --> 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.
```

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

Cevap:

b0 = 0 l0 = 8 b1 = 8 l1 = 8

Bu taban ve sınır değerleriyle, soruda belirtilen adres akışı şu şekilde çevrilir:

A0: geçerli (0 segmentine ait)
A1: geçerli (0 segmentine ait)
A2: ihlal (1. segmente ait, ancak 1. segment geçerli değil)
...
A15: geçerli (segment 1'e ait)
Segment 0'ın tabanı 0'a ayarlanır, bu, ilk 8 adresin (A0 - A7) segment 0'a ait olduğu anlamına gelir. Segment 0'ın uzunluğu 8'e ayarlanır, yani segment 0'daki 8 adresin tümü geçerlidir.

Segment 1'in tabanı 8'e ayarlanır, bu da son 8 adresin (A8 - A15) segment 1'e ait olduğu anlamına gelir. Segment 1'in uzunluğu 8'e ayarlanır, yani segment 1'deki 8 adresin tümü geçerlidir.

Bu taban ve sınır değerleriyle, soruda belirtilen adres akışı, açıklandığı gibi çeviri sonuçlarını üretecektir.

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 sonucu elde etmek için hangi parametreler önemlidir?

Cevap:

Rastgele oluşturulmuş sanal adreslerin kabaca %90'ının geçerli olmasıyla ilgili bir sorun oluşturmak için, simülatörü yeterince

fazlasıyla sayıda geçerli sanala adrese sahip olacak şekilde yapılandırmamız gerekir. Bu, sanal bellek alanındaki sayfa sayısını yeterince yüksek ayarlayarak elde edilebilir. Ek olakar, sayfa tablosunun düzgün bir şekilde ayarlandığından ve oluşturulan sanal adreslerin geçerli sanal adresler aralığında olduğundan emin olmanız gerekir.

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

Cevap:

Hiçbir sanal adresin geçerli olmadığı bir simulator çalıştırmak mümkün değildir. Sanal adresler, bilgisayarın işletim sistemi tarafından bilgisayarın belleğindeki belirli bellek konumlarını tanımlamak ve bunlara erişmek için kullanılır. Bu adresler, işletim sisteminin ve üzerinde çalışan programların düzgün çalışması için gereklidir, dolayısıyla geçerli bir sanal adres olmadan bir s,mülatörü çalıştırmak mümkün değildir.