16

Segmentasyon

Şimdiye kadar her **işlemin (process)** tüm adres alanını **belleğe (memory)** koyuyorduk. Temel ve sınır **kayıtlarıyla (registers)**, işletim sistemi işlemleri **fiziksel belleğin (physical memory)** farklı bölümlerine kolayca taşıyabilir. Bununla birlikte, bu adres alanlarımız hakkında ilginç bir şey fark etmiş olabilirsiniz : tam ortada, **yığın (stack)** ve yığın arasında büyük bir "boş" alan yığını var .

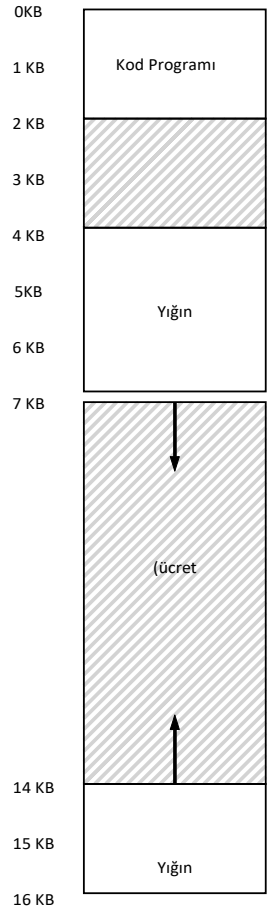
Şekil 16.1'den de tahmin edebileceğiniz gibi, yığın bir d yığını arasındaki boşluk işlem tarafından kullanılmamasına rağmen, tüm adres alanını fiziksel bellekte bir yere yerleştirdiğimizde hala fiziksel belleği kaplamaktadır; Bu nedenle, belleği sanallaştırmak için bir taban ve sınırlar kayıt çifti kullanmanın basit yaklaşımı israftır. Ayrıca, tüm adres alanı belleğe sığmadığında bir programı çalıştırmayı oldukça zorlaştırır; bu nedenle, taban ve sınırlar istediğimiz kadar esnek değildir. Ve böylece:

TO CRUX: HOW T Veya S YUKARI TAŞIMA A LARGE Acesaret SADIM

Yığın ve yığın arasında (potansiyel olarak) çok fazla boş alana sahip geniş bir adres alanını nasıl destekleriz? Örneklerimizde, küçük (taklit edilmiş) adres alanlarında, atıkların çok kötü görünmediğini unutmayın. Bununla birlikte, 32 bit adres alanı (4 GB benn boyutu); tipik bir program yalnızca megabaytlarca bellek kullanır, ancak yine de tüm adres alanının bellekte yerleşik olmasını gerektirir .

16.1 Segmentasyon: Genelleştirilmiş Taban/Sınırlar

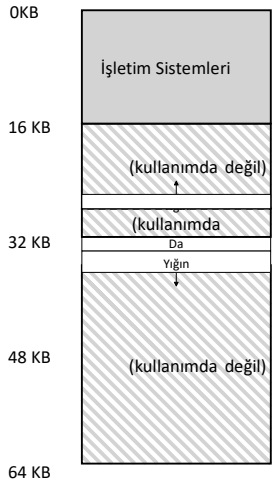
Bu sorunu çözmek için bir fikir doğdu ve buna **segmentasyon (segmentation)** adı **verildi**. Bu oldukça eski bir fikirdir, en azından 1960'ların başlarına kadar uzanır [H61, G62]. Fikir basit: MMU'muzda yalnızca bir taban ve sınır çiftine sahip olmak yerine, neden adres alanının mantıksal **segmenti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



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

adres alanı, mantıksal olarak farklı üç segmentimiz var: kod, yığın ve yığın. 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 üyeliğe yerleştirebiliriz. Örneğin, bkz: Şekil 16.2 (sayfa 3); orada, 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, yalnızca kullanılan bellek fiziksel bellekte yer ayırır ve böylece büyük miktarda kullanılmayan adres alanına (bazen **seyrek adres alanları (sparse address spaces)** diyoruz) sahip büyük adres alanları barındırılabilir.

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 sınavın kayıt değerlerini göstermektedir ; her sınır kaydı bir segmentin boyutunu tutar.

Segment	Taban	Boyutu
Kod	32K	2K
Yığın	34K	3K
Yığın	28K	2K

Şekil 16.3: **Segment Kayıt Değerleri**

Şekilden, kod segmentinin 32KB fiziksel adrese yerleştirildiğini ve 2KB boyutuna sahip olduğunu ve yığın 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, bir programın ne zaman yasadışı bir erişim yaptığını sabit yazılımın belirlemesini sağlar) bu sınırların dışında).

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 segmentindedir). Ne zaman referans-

BIRTARAF: TO SEGMENTASYON FAULT

Segmentasyon hatası (segmentation fault) veya ihlali terimi, segmentlere ayrılmış bir makinedeki bellek erişiminden yasadışı bir adrese kaynaklanır. Esprili bir şekilde, terim, segmentasyon desteği olmayan makinelerde bile devam ediyor. Zaten da çok esprili bir şekilde değil, kodunuzun neden arızalanmaya devam ettiğini anlayamıyorsanız.

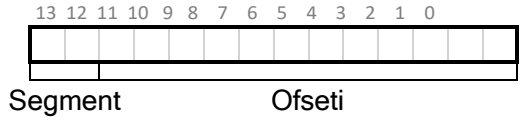
Zaman gerçekleşir (örneğin, bir talimat getirme işleminde), donanım istenen fiziksel adrese ulaşmak için *ofsete* temel değeri bu segmente (bu durumda 100) ekleyecektir : 100 + 32KB veya 32868. Daha sonra adresin sınırlar içinde olup olmadığını kontrol eder (100, 2 KB'den azdır), öyle olduğunu bulur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

Hayırw Yığındaki bir adrese bakalım, sanal adres 4200 (yine Şekil 16.1'e bakın). Yığının tabanına 4200 sanal adresini eklersek (34KB), doğru fiziksel adres olmayan 39016'lık bir fiziksel adres elde ederiz . İlk önce yapmamız gereken şey, *ofseti* yığına, yani *adresin bu segmentte* hangi baytlara atıfta bulunduğunu çıkarmaktır. Yığın sanal adres 4KB (4096) ile başladığından, 4200 uzaklığı aslında 4200 eksi 4096 veya 104'tür. Daha sonra bu ofseti (104) alırız ve istenen sonucu elde etmek için temel kayıt fiziksel adresine (34K) ekleriz: 34920.

Yığının sonunun ötesinde olan yasadışı bir adrese (yani, 7KB veya daha büyük bir sanal adres) atıfta bulunmaya çalışırsak ne olur? Ne olacağını hayal edebilirsiniz: donanım, adresin sınırların dışında olduğunu, işletim sistemine tuzaklar kurduğunu ve muhtemelen rahatsız edici işlemin sona ermesine yol açtığını tespit eder. Ve şimdi tüm C programerlerinin korkmayı öğrendiği ünlü terimin kökenini biliyorsunuz: **segmentasyon ihlali (segmentation violation)** veya **segmentasyon hatası (segmentation fault)**.

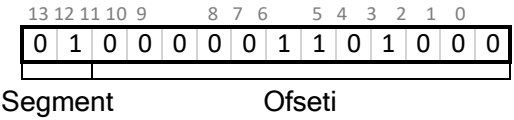
16.2 Hangi segmentten bahsediyoruz ?

Donanım, çeviri sırasında segment kayıtlarını kullanır . Bir segmente olan ofseti ve bir adresin hangi segmente atıfta bulunduğunu nasıl bilir? Bazen **açık (explicit)** bir yaklaşım olarak adlandırılan ortak bir yaklaşım, adres alanını sanal adresin en üstteki birkaç bitine göre bölümlere ayırmaktır ; 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 böylece yığın tabanını ve sınırlarını kullanır. Örneğimizi yukarıdan yığın sanal adresi (4200) alalım ve bunun açık olduğundan emin olmak için çevirelim. İkili formda 4200 sanal adresi burada görülebilir :



Resimden de görebileceğiniz gibi, üstteki iki bit (01) donanıma hangi *segmentten* bahsettiğimizi söyler. Alttaki 12 bit, segmente *ofsettir*: 0000 0110 1000 veya onaltılık 0x068 veya ondalık olarak 104. Bu nedenle, donanım hangi segment reg- ister'in kullanılacağını belirlemek için ilk iki b'yi alır ve ardından sonraki 12 biti segmente ofset olarak alır. Temel kaydı ofsete ekleyerek, donanım maddi fiziksel adrese ulaşır. Ofsetin sınır kontrolünü de kolaylaştırdığına dikkat edin: ofsetin sınırlardan daha az olup olmadığını kontrol edebiliriz; değilse, adres ille-gal'dir . Bu nedenle, taban ve sınırlar diziler olsaydı (segment başına bir girişle), donanım istenen fiziksel adresi elde etmek için aşağıdaki gibi bir şey yapıyor olurdu:

```
1 // 14 bit VA'nın en iyi 2 bitini edinin
2 Segment = (Sanal Adres ve SEG_MASK) >> SEG_SHIFT
3 // şimdi ofset olsun
4 Uzaklık = Sanal Adres & OFFSET_MASK
5 if (Ofset >= Bounds[Segment])
6       RaiseException(PROTECTION_FAULT)
7 başka
8       PhysAddr = Taban [Segment] + Ofset
9       Kayıt = AccessMemory(PhysAddr)
```

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

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 segment seçmek için üstteki bu kadar çok bitin kullanılmasıyla ilgili bir başka sorun, sanal adres alanının kullanımını sınırlamasıdır. Özellikle, her segment, örneğimizde 4 KB olan *maksimum* boyutla sınırlıdır (segmentleri seçmek için en üstteki iki biti kullanmak, 16 KB'lık adres alanının dört parçaya veya bu örnekte 4 KB'a bölündüğü anlamına gelir). Çalışan bir program bir segmenti (örneğin yığın veya yığın) bu maksimumun ötesine büyütmek isterse, programın şansı kalmaz.

Donanımın belirli bir adresin hangi kesimde olduğunu belirlemesinin başka yolları da vardır. **Örtük (implicit)** yaklaşımda, donanım caydırıcı-

Adresin nasıl oluştuğunu fark ederek segmenti mayınlar. Ex-ample için, adres program sayacından oluşturulduysa (yani, bir talimat getirme işlemiydi), adres kod segmenti içindedir; adres yığını veya temel işaretçiyi temel alıyorsa, yığın segmentinde olmalıdır; başka herhangi bir adres yığında olmalıdır.

16.3 Peki ya Yığın?

Şimdiye kadar, adres alanının önemli bir bileşenini dışarıda bıraktık: yığın. Yığın, yukarıdaki diyagramda 28 KB fiziksel adrese yeniden yerleştirildi, ancak kritik bir farkla: geriye doğru büyür (yani, daha düşük *adreslere doğru*). Fiziksel bellekte, 28KB 1'de "başlar" ve
16KB ila 14KB sanal adreslere karşılık gelen 26KB'a kadar büyür; çeviri farklı şekilde ilerlemelidir.

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

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

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

Segmentlerin ihmal edici yönde büyüyebileceği yönündeki donanım anlayışıyla, donanım artık bu tür sanal adresleri biraz farklı bir şekilde çevirmelidir. Bir örnek sanal adres yığını alalım ve süreci anlamak için trans- late edelim.

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). Sabit yazılım, segmenti belirlemek için en üstteki iki biti (11) kullanır, ancak daha sonra 3KB'lık bir ofset ile kalırız. Correct negatif ofsetini elde etmek için, maksimum segment boyutunu 3KB'den çıkarmalıyız: bu örnekte, bir ayrım 4KB olabilir ve bu nedenle doğru negatif ofset -1KB'ye eşit olan 3KB eksi 4KB'dir. 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] Basitlik için, yığının 28KB'de " başladığını" söylesek de, bu değer aslında 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 tasarımcıları kısa sürede biraz daha fazla donanım desteği ile yeni verimlilik türlerini gerçekleştirebileceklerini fark ettiler. Özellikle, bellekten tasarruf etmek için, bazen belirli bellek segmentlerini adres sp ace'leri arasında **paylaşmak (share)** yararlıdır. Özellikle, **kod paylaşımı (code sharing)** yaygındır ve günümüzde sistemlerde hala kullanılmaktadır.

Paylaşımı desteklemek için , donanımdan **koruma bitleri (protection bits)** şeklinde biraz daha fazla desteğe ihtiyacımız var. Temel destek, bir programın bir segmenti okuyup okuyamayacağını veya yazamayacağını veya belki de segmentin içinde yer alan kodu çalıştırıp çalıştıramayacağını gösteren segment başına birkaç bit ekler . Bir kod segmentini salt okunur olarak ayarlayarak, aynı kod, yalıtıma zarar verme endişesi olmadan birden çok işlem arasında paylaşılabilir ; Her işlem hala kendi özel belleğini bıraktığını düşünürken, işletim sistemi gizlice işlem tarafından değiştirilemeyen belleği paylaşıyor ve böylece illüzyon korunur.

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

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

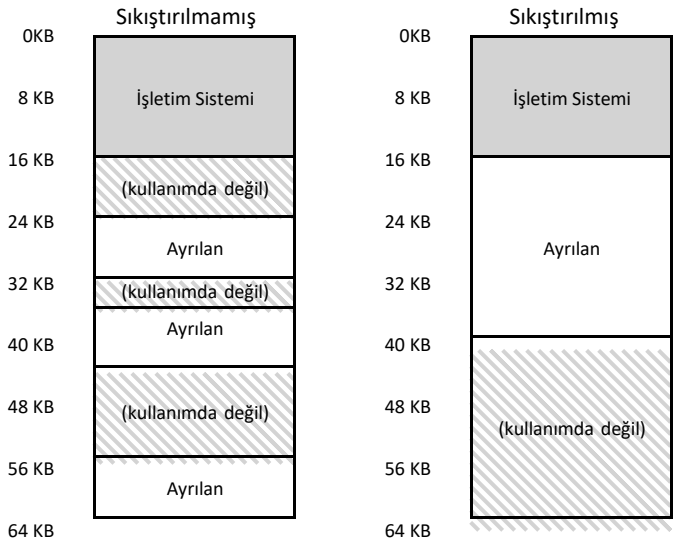
Şekil 16.5: **Segment Kayıt Değerleri (Korumalı)**

Koruma bitlerinde, daha önce açıklanan donanım algoritmasının da değişmesi gerekir. Sanal bir adresin sınırlar içinde olup olmadığını denetlemenin yanı sıra, donanımın belirli bir erişime izin verilip verilmediğini de denetlemesi gerekir. Bir kullanıcı işlemi salt okunur bir segmente yazmaya veya yürütülemez bir segmentten yürütmeye çalışırsa, donanım bir istisna oluşturmalı ve böylece işletim sisteminin rahatsız edici işlemle ilgilenmesine izin vermelidir.

16.5 İnce taneli vs Kaba taneli Segmentasyon

Şimdiye kadarki örneklerimizin çoğu, yalnızca birkaç segmente (yani kod, yığın, yığın) sahip sistemlere odaklanmıştır; Bu segmentasyonu **kaba taneli (coarse-grained)** olarak düşünebiliriz, çünkü adres alanını nispeten büyük, kaba parçalara böler. Bununla birlikte, bazı erken sistemler (örneğin, Multics [CV65, DD68]) daha esnekti ve adres alanlarının **ince taneli (fine-grained)** segmentasyon olarak adlandırılan çok sayıda küçük segmentten oluşmasına izin verdi.

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



Şekil 16.6: **Sıkıştırılmamış ve Sıkıştırılmış Bellek**

kod ve verileri , işletim sistemi ve donanımın daha sonra destekleyeceği ayrı segmentlere dönüştürür [RK68]. 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 olmalıdır. Adres alanının parçaları, sistem çalıştıkça fiziksel belleğe yeniden yerleştirilir ve böylece yalnızca tek bir taban / sınır çifti ile daha basit yaklaşımımıza göre büyük bir fiziksel bellek tasarrufu elde edilir . tüm adres alanı. Özellikle, yığın ve yığın arasındaki tüm kullanılmayan alanın fiziksel bellekte ayrılması gerekmez, bu da fiziksel belleğe daha fazla adres alanı sığdırmamıza ve büyük bir ve işlem başına seyrek sanal adres alanı.

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

Second, 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

TIP: BENF 1000 SOLUTION'lar VEcesaret, N Veya GREAT VeyaDEĞİL DYAŞ

Dışsal parçalanmayı içinde aza indirmeye çalışmak için bu kadar çok farklı algoritmanın nerede olması, daha güçlü bir temel gerçeğin göstergesidir: çözmenin "en iyi" yolu yoktur. sorun. Böylece, reasonable bir şeye razı oluruz ve yeterince iyi olmasını umarız. Tek gerçek çözüm (gelecek bölümlerde göreceğimiz gibi), belleği asla değişken boyutlu parçalara ayırmayarak sorunu tamamen önlemektir.

malloc() nesne için boş alan bulur ve arayana bir işaretçi döndürür . Bununla birlikte, diğerlerinde, yığın segmentinin kendisinin büyümesi gerekebilir. Bu durumda, bellek ayırma kitaplığı yığını büyütmek için bir sistem çağrısı gerçekleştirir (örneğin, geleneksel UNIX sbrk() sistem çağrısı). İşletim sistemi daha sonra (genellikle) daha fazla alan sağlar, segment boyutu reg- ister'i yeni (daha büyük) boyuta günceller ve başarı kitaplığını bilgilendirir; Kütüphane daha sonra yeni nesne için alan 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 bir dizi segmentimiz var ve her segment farklı bir boyutta olabilir .

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

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

Bu sorunun bir çözümü, mevcut segmentleri yeniden düzenleyerek fiziksel belleği **sıkıştırmak (compact)** olacaktır. Örneğin, işletim sistemi hangi işlemlerin çalıştığını durdurabilir, verilerini bitişik bir bellek yenisine kopyalayabilir , segment kayıt değerlerini yeni fiziksel konumlara işaret edecek şekilde değiştirebilir ve böylece hangisi işe yarayacak. Bunu yaparak, işletim sistemi yeni ayırma isteğinin başarılı olmasını sağlar. Bununla birlikte, kopyalama segmentleri bellek yoğun olduğundan ve genellikle adil miktarda işlemci süresi kullandığından sıkıştırma pahalıdır; görmek

Sıkıştırılmış fiziksel belleğin diyagramı için Şekil 16.6 (sağda). Com-paction ayrıca (ironik bir şekilde) mevcut segmentleri büyütme taleplerinin karşılanmasını zorlaştırır ve bu nedenle bu tür talepleri karşılamak için daha fazla yeniden düzenlemeye neden olabilir.

Bunun yerine, daha basit bir yaklaşım, büyük miktarda belleği tahsis için kullanılabilir tutmaya çalışan bir serbest liste yönetim algoritması kullanmak olabilir. Kelimenin tam anlamıyla, **en uygun (best-fit)** gibi klasik algoritmalar da dahil olmak üzere insanların aldığı yüzlerce yaklaşım vardır (boş alanların bir listesini tutar ve istenen şeyi karşılayan boyut olarak en yakın olanı döndürür). talepte bulunana tahsis), **en kötü uyum (worst-fit)**, **ilk uyum (first-fit)** ve **arkadaş algoritması (buddy algorithm)** [K68] gibi daha karmaşık şemalar. Wilson ve ark. 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 hala 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 sanallaştırmasını oluşturmamıza yardımcı olur. Yalnızca dinamik yer değiştirmenin ötesinde, ayrım, adres alanının mantıksal bölümleri arasındaki büyük potansiyel bellek israfını önleyerek seyrek adres alanlarını daha iyi destekleyebilir. Aynı zamanda hızlıdır, çünkü aritmetik segmentasyonun gerektirdiği gibi kolay ve har dware'e çok uygundur; çevirinin ek yükleri minimumdur. Bir yan fayda da ortaya çıkıyor: kod paylaşımı. Kod ayrı bir segment içine yerleştirilirse , böyle bir segment potansiyel olarak birden çok çalışan program arasında paylaşılabilir .

Bununla birlikte, öğrendiğimiz gibi, değişken boyutlu segmentleri toplantıya 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 bellek ayırma isteğini yerine getirmek zor olabilir. Akıllı algoritmalar [W+95] veya periyodik olarak kompakt bellek kullanılmaya çalışılabilir, ancak sorun temel ve kaçınılması zordur.

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

Başvuru

[CV65] "Multics Sistemine Giriş ve Genel Bakış" F. J. Corbato, v. Bir. Vyssotsky. Güz Ortak Bilgisayar Konferansı, 1965. *Güz Ortak Bilgisayar Konferansı'nda Multics hakkında sunulan beş bildiriden biri ; oh o gün o odada duvarda bir sinek olmak*!

[DD68] "Virtual Memory, Processes, and Sharing in Multics" Robert C. Daley ve Jack B. Dennis tarafından yazılmıştır. ACM'nin İletişimi, Cilt 11:5, Mayıs 1968. *Multics'te dinamik bağlantının nasıl gerçekleştirileceğine dair erken bir makale, zamanının çok ilerisindeydi. Dinamik linkin* g, *büyük X-windows kütüphanelerinin talep ettiği gibi, yaklaşık 20 yıl sonra sistemlere geri döndü. Bazıları, bu büyük X11 kütüphanelerinin,* U *NIX'in ilk sürümlerinde dinamik bağlantı desteğini kaldırdığı için MIT'nin intikamı olduğunu söylüyor*!

[G62] M. tarafından "Gerçek Segmentasyonu". N. Greenfield. SJCC Bildirileri, Cilt 21, Mayıs 1962. *Segmentasyon üzerine bir başka erken makale ; o kadar erken ki, diğer çalışmalara atıfta bulunmuyor.*

[H61] "Dinamik Depolama için Program Organizasyonu ve Kayıt Tutma", A. W. Holt. ACM'nin Komünikasyonları, Cilt 4:10, Ekim 1961. *Segmentasyon* ve *bazı kullanımları hakkında inanılmaz derecede erken ve okunması zor bir makale.*

[I09] Intel tarafından "Intel 64 ve IA-32 Mimarileri Yazılım Geliştirici El Kitapları". 2009. Kullanılabilir: http://www.intel.com/products/processor/manuals. *Burada segmentasyon hakkında tekrar eklemeyi deneyin (Cilt 3a'da* Bölüm *3); başınızı acıtacak, en azından birazcık.*

[K68] " Bilgisayar Programlama Sanatı: Cilt I", Donald Knuth. Addison-Wesley, 1968. *Knuth sadece Bilgisayar Programlama Sanatı hakkındaki ilk kitaplarıyla değil, aynı zamanda bugün hala profesyoneller tarafından kullanılan bir güç merkezi dizgi aracı olan dizgi sistemi TeX* ve *aslında bu kitabı dizgilemek için ünlüdür.* Algoritmalar hakkındaki metinleri, *bugün bilgisayar sistemlerinin altında yatan algoritmaların çoğuna büyük bir erken referanstır.*

[L83] Butler Lampson tarafından "Bilgisayar Sistemleri Tasarımı için İpuçları ". ACM İşletim Sistemleri İncelemesi, 15:5, Ekim 1983. *Sistemlerin nasıl inşa edileceğine dair bilge tavsiyelerinin hazinesi. Bir oturuşta okunması zor ; iyi* bir *şarap veya referans kılavuzu gibi bir seferde biraz alın.*

[LL82] "VAX/VMS İşletim Sisteminde Sanal Bellek Yönetimi", Henry M. Levy, Peter H. Lipman. IEEE Computer, Cilt 15:3, Mart 1982. *Tasarımında çok fazla sağduyu bulunan klasik bir bellek yönetim sistemi. Daha sonraki bir* bölümde *daha ayrıntılı olarak inceleyeceğiz .*

[RK68] "Dinamik Depolama Tahsis Sistemleri" B. Randell ve C.J. Kuehner. ACM'nin Tebliğleri, Cilt 11:5, Mayıs 1968. *Sayfalama ve segmentasyon arasındaki farklara güzel bir genel bakış, çeşitli* makinelerin *bazı tarihsel tartışmalarıyla.*

[R69] Brian Randell tarafından "Depolama parçalanması ve program segmentasyonu üzerine bir not". ACM'nin Komünikasyonları, Cilt 12:7, Temmuz 1969. *Parçalanmayı tartışmak için en eski makalelerden* biri.

[G+95] "Dinamik Depolama Tahsisi: Bir Anket ve Eleştirel İnceleme", Paul R. Wilson, Fransa S. Johnstone, Michael Neely, David Boles. Uluslararası Bellek Yönetimi Çalıştayı, İskoçya, İngiltere, Eylül 1995. *Bellek ayırıcılar hakkında harika bir anket makalesi.*

Ödev (Simülasyon)

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

Soru

1. İlk önce bazı adresleri çevirmek için küçük bir adres alanı kullanalım. İşte birkaç farklı rastgele tohum 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
```

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?

3. 128 baytlık bir fiziksel bellekte 16 baytlık küçük bir adres alanımız olduğunu varsayalım. Simülatörün belirtilen adres akışı için aşağıdaki çeviri sonuçlarını oluşturmasını sağlamak için hangi temel ve sınırları ayarlarsınız: geçerli, geçerli, ihlal, ..., violation, geçerli , geçerli? 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 ?
```

Belirtilen adres akışı için belirtilen parametrelerle istenen çeviri sonuçlarını elde etmek için, her bölüm için ayarlanacak taban ve sınırların belirlenmesi gerekir. Burada bir çalışacak olan ayarlanma yapısı:
-b0 0 - -l0 16
-b1 16 - -l1 16
Bu yapılandırmada, ilk bölüm tüm 16 baytlık adres alanını kapsayacak ve ikinci bölüm geri kalan 112 baytı fiziksel bellekte kapsayacaktır.
Aşağıdaki kod, verilen parametreler için segmentasyon yapılandırmasının nasıl uygulanabileceğini gösterir:

```python
# Parametreleri tanımla
adres_alanı_boyutu = 16
fiziksel_bellek_boyutu = 128

# Adres akışını tanımla
adres_akışı = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

# Bölümleri ayarla
segment1_tabanı = 0
segment1_uzunluğu = 16
segment2_tabanı = 16
segment2_uzunluğu = 16

# Adres akışını döngü ile geç ve geçerli/geçersiz çevirileri kontrol et
for adres in adres_akışı:
  if adres < adres_alanı_boyutu:
    # Adres adres alanı içinde, fiziksel bellekte geçerli bir konuma eşlenip eşlenmediğini kontrol et
    if adres < segment1_uzunluğu:
      # Adres segment 1'e eşlenmiş, segment 1 sınırları içinde mi kontrol et
      if adres >= segment1_tabanı and adres < segment1_tabanı + segment1_uzunluğu:
        # Geçerli çeviri
        print("Adres {} için geçerli çeviri".format(adres))
      else:
        # Geçersiz çeviri
        print("Adres {} için geçersiz çeviri".format(adres))
    else:
      # Adres segment 2'ye eşlenmiş, segment 2 sınırları içinde mi kontrol et
      if adres >= segment2_tabanı and adres < segment2_tabanı + segment2_uzunluğu:
        # Geçerli çeviri
        print("Adres {} için geçerli çeviri".format(adres))
      else:
        # Geçersiz çeviri
        print("Adres {} için geçersiz çeviri".format(adres))
  else:
    # Adres adres alanı dışında, geçersiz çeviri
    print("Adres {} için geçersiz çeviri".format(adres))
```

```
Adres 0 için geçerli çeviri
Adres 1 için geçerli çeviri
Adres 2 için geçersiz çeviri
Adres 3 için geçersiz çeviri
Adres 4 için geçersiz çeviri
Adres 5 için geçersiz çeviri
Adres 6 için geçersiz çeviri
Adres 7 için geçersiz çeviri
Adres 8 için geçersiz çeviri
Adres 9 için geçersiz çeviri
Adres 10 için geçersiz çeviri
Adres 11 için geçersiz çeviri
Adres 12 için geçersiz çeviri
Adres 13 için geçersiz çeviri
Adres 14 için geçersiz çeviri
Adres 15 için geçerli çeviri
```

Bu kod, verilen adres akışı için çeviri sonuçlarına uyan yandaki çıktıyı üretir:

4. Rastgele oluşturulan 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?

Sanal adreslerin geçerlilik oranını ayarlamak için, simülatörün bellek bölümleme ayarlarını değiştirmemiz gerekebilir. Sanal adreslerin geçerlilik oranını ayarlamak için geçerli parametreler şunlardır:

- Bellek bölümlerinin boyutları ve sayısı
- Bellek bölümlerinin hangi adres aralıklarında atandığı

Bu parametreleri değiştirerek geçerli adreslerin oranını ayarlayabiliriz.

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

Sanal adres geçerli olmayacak şekilde çalıştırılabilir.

Sanal adresleri devre dışı bırakmak, bir simülatörün doğru çalışması için gerekli olmayabilir. Ancak, bu tür bir işlem yapılırsa, simülatörün performansı ve doğruluk ihtimali düşebilir. Eğer sanal adreslerin kullanılması gerekli değilse ve sadece simülatörün çalıştırılması isteniyorsa, bu işlemi yapmak için aşağıdaki adımları izlenir:

1. İlk olarak, simülatörün kurulu olduğu bilgisayarın işletim sistemini açarız.

2. Daha sonra, simülatörün yüklü olduğu klasöre gidin ve simülatörün ayar dosyasını buluruz. Bu dosya genellikle "settings" veya "config" gibi bir isimle kaydedilir ve simülatörün kurulu olduğu klasöre kaydedilir.

3. Simülatörün ayar dosyasını açın ve içindeki sanal adresleri devre dışı bırakacak bir seçeneği buluruz. Bu seçenek genellikle "disable virtual addressing" veya "disable virtual memory" gibi bir ifadeyle belirtilir ve ayar dosyasının en altında bulunur.

4. Sanal adresleri devre dışı bırakacak olan seçeneği aktif hale getiririz ve ayar dosyasını kaydederiz.

5. Son olarak, simülatörü tekrar başlatırız ve sanal adreslerin devre dışı bırakılmış olduğunu kontrol ederiz.

Bu adımları izleyerek sanal adresleri devre dışı bırakabiliriz.