18

Sayfalama: Giriş

Bazen işletim sisteminin herhangi bir alan yönetimi sorununu çözerken iki yaklaşımdan birini kullandığı söylenir.parçalara ayırmaktır *değişken boyutlu* gördüğümüz gibi, **segmentasyonda(segmantation)** sanal bellekteNe yazık ki, bu çözümün kendine özgü güçlükleri var. Özellikle, bir alanı farklı büyüklükteki parçalara bölerken, alanın kendisi parçalanabilir ve bu nedenle, ayırma zamanla daha zor hale gelir. Parçalara bölmek *sabit boyutlu* . Sanal bellekte bu fikre **sayfalama(paging)** ve bu fikir erken ve önemli bir sisteme, Atlas'a [KE+62, L78] kadar uzanıyor. Bir sürecin adres uzayını değişken boyutlu mantıksal parçalara (örn. kod, öbek, yığın) bölmek yerine, onu her birine **sayfa(page)** adını verdiğimiz sabit boyutlu birimlere ayırırız.adı verilen sabit boyutlu yuvalar dizisi olarak görüyoruz **sayfa çerçeveleri(page frames)**; bu çerçevelerin her biri tek bir sanal bellek sayfası içerebilir. Karşılaştığımız

Önemli Nokta:

BELLEĞİ SAYFALARLA NASIL SANALLAŞTIRIRSINIZ?

Bölümleme sorunlarından kaçınmak için belleği sayfalarla nasıl sanallaştırabiliriz? Temel teknikler nelerdir? Bu teknikleri minimum yer ve zaman giderleri ile nasıl iyi çalışır hale getirebiliriz

18.1 Basit Bir Örnek ve Genel Bakış

Bu yaklaşımı daha açık hale getirmek için basit bir örnekle açıklayalım. Şekil 18.1 (sayfa 2), dört adet 16 bayt sayfalı (sanal sayfalar 0, 1, 2 ve 3) toplam boyutu yalnızca 64 bayt olan küçük bir adres alanı örneği sunar. Gerçek adres alanları çok daha büyüktür, tabii ki genellikle 32 bit ve dolayısıyla 4 GB adres alanı, hatta 64 bit[1]; kitapta, sindirilmelerini kolaylaştırmak için genellikle küçük örnekler kullanacağız

[1]A 64-bit address space is hard to imagine, it is so amazingly large. An analogy might help: if you think of a 32-bit address space as the size of a tennis court, a 64-bit address space is about the size of Europe(!).

Şekil 18.1: **Basit 64 baytlık Adres Alanı**

Fiziksel bellek, Şekil 18.2'de gösterildiği gibi , ayrıca bir dizi sabit boyutlu yuvadan oluşur, bu durumda sekiz sayfa çerçevesi (128 baytlık bir fiziksel bellek oluşturur, yine gülünç derecede küçüktür). Diyagramda görebileceğiniz gibi, sanal adres alanının sayfaları fiziksel bellek boyunca farklı konumlara yerleştirilmiştir; diyagram ayrıca işletim sisteminin fiziksel belleğin bir kısmını kendisi için kullandığını gösterir.

Göreceğimiz gibi sayfalama, önceki yaklaşımlarımıza göre bir takım avantajlara sahiptir. Muhtemelen en önemli gelişme *esneklik* olacaktır: tamamen geliştirilmiş bir sayfalama yaklaşımıyla sistem, bir işlemin adres alanını nasıl kullandığından bağımsız olarak, bir adres alanının soyutlanmasını etkili bir şekilde destekleyebilecektir; örneğin, yığının ve yığının büyüme yönü ve bunların nasıl kullanıldığı hakkında varsayımlarda bulunmayacağız.

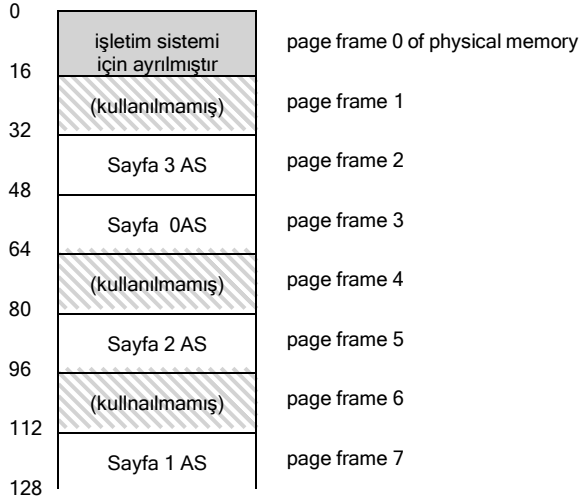


Figure 18.2: **128-baytlık Fiziksel Bellekte 64-Baytlık adres alanı**

Diğer bir avantaj da, *basitliğidir* sağladığı boş alan yönetiminin Örneğin, işletim sistemi 64 baytlık küçük adres alanımızı sekiz sayfalık fiziksel belleğimize yerleştirmek istediğinde, yalnızca dört boş sayfa bulur; belki işletim sistemi **ücretsiz bir listesini(free list)** ve bu listeden ilk dört boş sayfayı alır. Örnekte, işletim sistemi adres alanının (AS) sanal 0. sayfasını fiziksel çerçeve 3'e, AS'nin sanal 1. sayfasını fiziksel çerçeve 7'ye, sayfa 2'yi çerçeve 5'e ve sayfa 3'ü çerçeve 2'ye yerleştirmiştir. Sayfa çerçeveleri 1,4 ve 6 şu anda ücretsiz.

Adres alanının her bir sanal sayfasının fiziksel bellekte nereye yerleştirildiğini kaydetmek için, işletim sistemi genellikle *işlem başına* veri yapısı **sayfa tablosu(page table)** .depolamak **adres çevirilerini(address translations)**, adres uzayındaki sanal sayfaların her biri içinBasit örneğimiz için (Şekil 18.2, sayfa 2), sayfa tablosunda şu dört giriş olacaktır: (Sanal Sayfa 0 → Fiziksel Çerçeve 3), (VP 1 → PF 7), (VP 2 → PF 5) ve (VP 3 → PF 2).

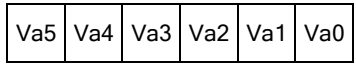
Bir veri yapısı olduğunu unutmamak önemlidir *işlem başına* (tartıştığımız sayfa tablosu yapılarının çoğu işlem başına yapılardır; üzerinde duracağımız bir istisna, tersine **çevrilmiş sayfa tablosudur(inverted page table)**). Yukarıdaki örneğimizde başka bir işlem çalışacak olsaydı, sanal sayfaları açıkça *farklı* fiziksel sayfalarla eşleştiğinden (herhangi bir paylaşım modulo) OS'nin bunun için farklı bir sayfa tablosu yönetmesi gerekirdi.

Artık bir adres çevirisi örneği yapacak kadar bilgimiz var. Bu küçücük adres alanıyla (64 bayt) işlemin bir bellek erişimi oluşturduğunu düşünelim:

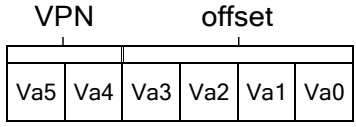
```
movl <sanal adres>, %eax
```

Spesifik olarak, <sanal adres> adresinden eax yazmacına açık veri yüküne dikkat edelim. (ve böylece daha önce gerçekleşmiş olması gereken komut getirme işlemini göz ardı edin).

için **çevirmek** önce onu iki bileşene ayırmamız gerekir: **sanal sayfa numarası (Virtuel page number,VPN)** ve **ofset** sayfa içindeki Bu örnek için, işlemin sanal adres alanı 64 bayt olduğu için, sanal adresimiz için toplam 6 bit ihtiyacımız var (2^6 = 64). Böylece, sanal adresimiz şu şekilde kavramsallaştırılabilir:



Bu diyagramda, Va5 sanal adresin en yüksek dereceli bitidir ve Va0 en düşük dereceli bittir. Sayfa boyutunu (16 bayt) bildiğimiz için, sanal adresi aşağıdaki gibi bölebiliriz:

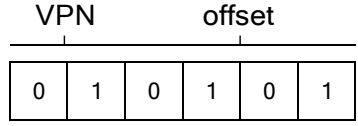


Sayfa boyutu 64 baytlık bir adres alanında 16 bayttır; bu nedenle 4 sayfa seçebilmemiz gerekir ve adresin en üst 2 biti tam da bunu yapar. Böylece 2 bitlik bir sanal sayfa numarasına (VPN) sahip oluruz. Kalan bitler bize sayfanın hangi baytıyla ilgilendiğimizi söyler, bu durumda 4 bit; buna ofset diyoruz.

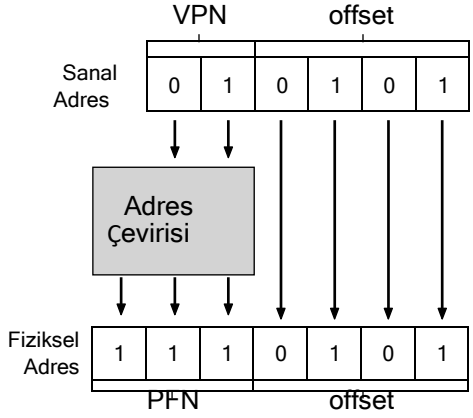
Bir süreç sanal bir adres ürettiğinde, işletim sistemi ve donanımın bunu anlamlı bir fiziksel adrese çevirmek için bir araya gelmesi gerekir. Örnek olarak, yukarıdaki yüklemenin 21 numaralı sanal adrese yapıldığını varsayalım:

```
movl 21, %eax
```

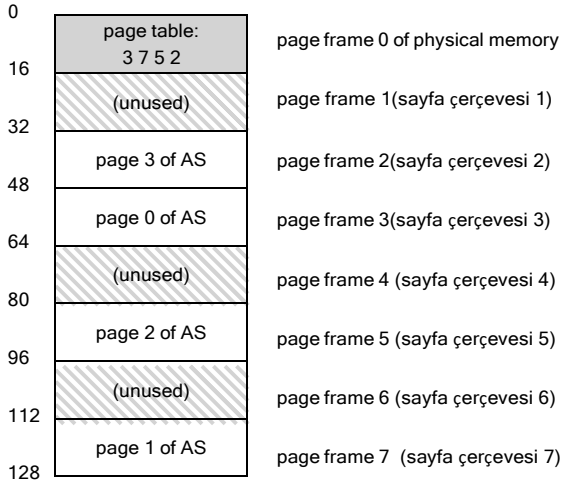
"21 "i ikili forma çevirdiğimizde "010101" elde ederiz ve böylece bu sanal adresi inceleyebilir ve sanal sayfa numarasına (VPN) ve ofsete nasıl ayrıldığını görebiliriz:



Böylece, "21" sanal adresi "01" (ya da 1) sanal sayfasının 5. ("0101 "inci) baytında yer alır. Sanal sayfa numaramızla artık sayfa tablomuzu indeksleyebilir ve sanal sayfa 1'in hangi fiziksel çerçeve içinde yer aldığını bulabiliriz. Yukarıdaki sayfa tablosunda **fiziksel çerçeve numarası (Physical Frame Number,PFN**) (bazen **fiziksel sayfa numarası(Physical Page Number)** ya da **PPN** olarak da adlandırılır) 7'dir (binary 111). Böylece, VPN'i PFN ile değiştirerek bu sanal adresi çevirebilir ve ardından fiziksel belleğe yükleme yapabiliriz (Şekil 18.3).



Şekil 18.3: **Adres Çeviri Süreci**



Şekil 18.4: **Örnek: Çekirdek Fiziksel Belleğindeki Sayfa Tablosu**

Ofsetin aynı kaldığına (yani çevrilmediğine) dikkat edin, çünkü ofset bize sadece sayfa *içinde* hangi baytı istediğimizi söyler. Son fiziksel adresimiz 1110101'dir (ondalık olarak 117) ve tam olarak yüklememizin veri almasını istediğimiz yerdir (Şekil 18.2, sayfa 2).

Bu temel genel bakışı akılda tutarak, şimdi sayfalama hakkında aklınıza gelebilecek birkaç temel soruyu sorabiliriz (ve umarız yanıtlayabiliriz). Örneğin, bu sayfa tabloları nerede saklanır? Sayfa tablosunun tipik içeriği nedir ve tablolar ne kadar büyüktür? Sayfalama sistemi (çok) yavaşlatır mı? Bunlar ve diğer merak uyandıran sorular, en azından kısmen, aşağıdaki metinde yanıtlanmaktadır. Okumaya devam edin!

18.2 Sayfa Tabloları Nerede Saklanır?

Sayfa tabloları çok büyük olabilir, daha önce tartıştığımız küçük segment tablosundan veya taban/sınırlar çiftinden çok daha büyük olabilir. Örneğin, 4KB sayfaları olan tipik bir 32-bit adres alanı düşünün. Bu sanal adres 20 bitlik VPN ve 12 bitlik ofsete bölünür (1KB sayfa boyutu için 10 bit gerektiğini hatırlayın ve 4KB'a ulaşmak için sadece iki tane daha ekleyin).

20 bitlik bir VPN, işletim sisteminin her işlem için yönetmesi gereken 2^{20} çeviri olduğu anlamına gelir (bu kabaca bir milyon eder); fiziksel çeviriyi ve diğer yararlı şeyleri tutmak için **sayfa tablosu girişi (Page table entry,PTE)** başına 4 bayta ihtiyacımız olduğunu varsayarsak, her sayfa tablosu için 4 MB'lık muazzam bir bellek gerekir! Bu oldukça büyük bir rakam. Şimdi 100 işlemin çalıştığını düşünün: bu, işletim sisteminin sadece tümbu adres çevirileri için 400MB belleğe ihtiyaç duyacağı anlamına gelir

Modern çağda bile, nerede

ASIDE: VERI YAPISI - SAYFA TABLOSU

Modern bir işletim sisteminin bellek yönetimi alt sistemindeki en önemli veri yapılarından biri **sayfa tablosudur (page table)**. Genel olarak, bir sayfa tablosu **sanal-fiziksel adres çevirilerini saklar (virtual to physical address translations),** böylece sistemin bir adres alanının her bir sayfasının fiziksel bellekte gerçekte nerede bulunduğunu bilmesini sağlar. Her adres alanı bu tür çeviriler gerektirdiğinden, genel olarak sistemdeki her işlem için bir sayfa tablosu vardır. Sayfa tablosunun tam yapısı ya donanım tarafından belirlenir (eski sistemler) ya da işletim sistemi tarafından daha esnek bir şekilde yönetilebilir (modern sistemler).

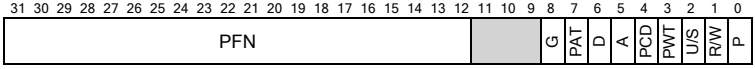
Makinelerin gigabaytlarca belleği varken, bunun büyük bir kısmını sadece çeviriler için kullanmak biraz çılgınca görünüyor, değil mi? Ve64 bitlik bir adres alanı için böyle bir sayfa tablosunun ne kadar büyük olacağını düşünmeyeceğiz bile; bu çok korkunç olurdu ve belki de sizi tamamen korkuturdu.

Sayfa tabloları çok büyük olduğundan, MMU'da o anda çalışmakta olan sürecin sayfa tablosunu saklamak için özel bir yonga üstü donanımtutmuyoruz. Bunun yerine, her işlem için sayfa tablosunu *bellekte bir* yerde saklarız. Şimdilik sayfa tablolarının işletim sisteminin yönettiği fiziksel bellekte bulunduğunu varsayalım; daha sonra işletim sistemi belleğinin büyük bir kısmının sanal hale getirilebileceğini ve böylece sayfa tablolarının işletim sistemi sanal belleğinde saklanabileceğini (ve hatta diske takas edilebileceğini) göreceğiz, ancak şu anda bu çok kafa karıştırıcı, bu yüzden bunu göz ardı edeceğiz. Şekil 18.4 (sayfa 5) OS belleğindeki bir sayfa tablosunun resmidir; oradaki küçük çeviri setini görüyor musunuz?

18.3 Sayfa Tablosunda Aslında Ne Var?

Sayfa tablosu organizasyonu hakkında biraz konuşalım. Sayfatablosu, sanal adresleri (ya da gerçekte sanal sayfa numaralarını) fiziksel adreslere (fiziksel çerçeve numaraları) eşlemek için kullanılan bir veri yapısıdır. Bu nedenle, herhangi bir veri yapısı çalışabilir. En basit biçimi **doğrusal sayfa tablosu (linear page table)** olarak adlandırılır ve bu sadece birdizidir. İşletim sistemi diziyi sanal sayfa numarasına (VPN) göre *indeksler* ve istenen fiziksel çerçeve numarasını (PFN) bulmak için bu indeksteki sayfa tablosu girişini (PTE) arar. Şimdilik, bu basit doğrusalyapıyı varsayacağız; daha sonraki bölümlerde, sayfalama ile ilgili bazı sorunları çözmeye yardımcı olmak için daha gelişmiş veri yapılarını kullanacağız.

Her PTE'nin içeriğine gelince, orada belli bir seviyede anlamaya değer bir dizi farklı bitimiz vardır. Geçerli bir **bit (valid bit),** belirli bir çevirinin geçerli olup olmadığını belirtmek için yaygın olarak kullanılır; örneğin, bir program çalışmaya başladığında, adres alanının bir ucunda kod ve yığın, diğer ucunda ise yığın olacaktır. Aradaki kullanılmayan tüm alan **geçersiz (invalid)** olarak işaretlenir ve süreç bu belleğe erişmeye çalışırsa, işletim sistemine bir tuzak oluşturacak ve muhtemelen süreci sonlandıracaktır. Bu nedenle, geçerli bit seyrek bir adres alanını desteklemek için çok önemlidir; adres alanındaki kullanılmayan tüm sayfaları geçersiz olarak işaretleyerek, bu sayfalar için fiziksel çerçeve ayırma ihtiyacını ortadankaldırır ve böylece büyük miktarda bellek tasarrufu sağlarız.



Şekil 18.5: **Bir x86 Sayfa Tablosu Girişi (PTE)**

Ayrıca sayfanın okunup okunamayacağını, yazılıp yazılamayacağını ya da çalıştırılıp çalıştırılamayacağını gösteren **koruma bitlerimiz (protection bits)** de olabilir. Yine, bu bitlerin izin vermediği bir şekilde bir sayfaya erişmek işletim sistemine bir tuzak oluşturacaktır.

Önemli olan birkaç bit daha vardır ancak şimdilik bunlardan fazla bahsetmeyeceğiz. **Present biti(present bit)** bu sayfanın fiziksel bellekte mi yoksa diskte mi olduğunu (yani **takas** edildiğini) gösterir. Fiziksel bellekten daha büyük adres alanlarını desteklemek için adres alanının bazı bölümlerinin diske nasıl **takas(swap)** edileceğini incelerken bu mekanizmayı daha iyi anlayacağız; takas, işletim sisteminin nadiren kullanılan sayfaları diske taşıyarak fiziksel belleği boşaltmasını sağlar. Sayfanın belleğe alındığından beri değiştirilip değiştirilmediğini gösteren **kirli bit(dirty bit)** de yaygındır.

Bir **referans biti** (diğer adıyla **erişilen bit(accessed bit)**) bazen bir sayfaya erişilip erişilmediğini izlemek için kullanılır ve hangi sayfaların popüler olduğunu ve dolayısıyla bellekte tutulması gerektiğini belirlemede yararlıdır; bu tür bir bilgi, sonraki bölümlerde ayrıntılı olarak inceleyeceğimiz bir konu olan **sayfa değiştirme(page replacement)** sırasında kritik öneme sahiptir.

Şekil 18.5'te x86 mimarisinden örnek bir sayfa tablosu girişi gösterilmektedir [I09]. Bir mevcut biti (P); bu sayfaya yazmaya izin verilip verilmediğini belirleyen bir okuma/yazma biti (R/W); kullanıcı modu işlemlerinin sayfaya erişip erişemeyeceğini belirleyen bir kullanıcı/denetmen biti (U/S); bu sayfalar için donanım önbelleğinin nasıl çalıştığını belirleyen birkaç bit (PWT, PCD, PAT ve G); bir erişilen bit (A) ve bir kirli bit (D) ve son olarak sayfa çerçeve numarasının (PFN) kendisini içerir.

X86 sayfa desteği hakkında daha fazla ayrıntı için Intel Mimarisi Kılavuzlarını [I09] okuyun. Ancak önceden uyaralım; bu gibi kılavuzları okumak oldukça bilgilendirici olsa da (ve işletim sisteminde bu tür sayfa tablolarını kullanmak için kod yazanlar için kesinlikle gerekli olsa da) ilk başta zorlayıcı olabilir. Biraz sabır ve çok fazla istek gereklidir.

BIR KENARA: NEDEN GEÇERLI BIT YOK?

Intel örneğinde, geçerli ve mevcut bitlerin ayrı olmadığını, bunun yerine sadece bir mevcut bit (P) olduğunu fark edebilirsiniz. Bu bitayarlanmışsa (P=1), sayfanın hem mevcut hem de geçerli olduğu anlamına gelir. Değilse (P=0), sayfanın bellekte mevcut olmayabileceği (ancak geçerli olduğu) veya geçerli olmayabileceği anlamına gelir. P=0 ile bir sayfaya erişim işletim sistemine bir tuzak tetikleyecektir; işletim sistemi daha sonra sayfanın geçerli olup olmadığını (ve bu nedenle belki de geri takas edilmesi gerektiğini) veya olmadığını (ve bu nedenle programın belleğe yasadışı olarak erişmeye çalıştığını) belirlemek için tuttuğu ek yapıları kullanmalıdır. Bu tür bir ihtiyatlılık, genellikle işletim sisteminin tam bir hizmet oluşturabileceği asgari özellikler

18.4 Çağrı: Ayrıca Çok Yavaş

Bellekteki sayfa tablolarının çok büyük olabileceğini zaten biliyoruz. Görünüşe göre, işleri de yavaşlatabiliyorlar. Örneğin, basit talimatımızı ele alalım:

```
movl 21, %eax
```

Yine, sadece 21 numaralı adrese yapılan açık referansı inceleyelim ve komut getirme konusunda endişelenmeyelim. Bu örnekte, donanımın bizim için çeviriyi gerçekleştirdiğini varsayacağız. İstenen veriyi almak için, sistem önce sanal adresi (21) doğru fiziksel adrese (117) **çevirmelidir(translate).** Böylece, 117 adresinden veriyi almadan önce, sistem önce işlemin sayfa tablosundan uygun sayfa tablosu girişini almalı, çeviriyi gerçekleştirmeli ve ardından veriyi fiziksel bellekten yüklemelidir.

Bunu yapmak için, donanımın o anda çalışan işlem için sayfa tablosunun nerede olduğunu bilmesi gerekir. Şimdilik tek bir **sayfa tablosu temel kaydının (page table base register)** sayfa tablosunun başlangıç konumunun fiziksel adresini içerdiğini varsayalım. İstenen PTE'nin yerini bulmak için donanım aşağıdaki işlevleri yerine getirecektir:

```
VPN= (VirtualAddress & VPN_MASK) >> SHIFT PTEAddr
= PageTableBaseRegister + (VPN * sizeof (PTE))
```

Örneğimizde, `VPN MASK` 0x30 (hex 30 veya binary 110000) olarak ayarlanır, bu da VPN bitlerini tam sanal adresten seçer; `SHIFT` 4'e (ofsetteki bit sayısı) ayarlanır, böylece VPN bitlerini doğru tamsayı sanal sayfa numarasını oluşturmak için aşağı taşırız. Örneğin, sanaladres 21 (010101) ve maskeleme bu değeri 010000'e dönüştürür; kaydırma, istenildiği gibi 01'e veya sanal sayfa 1'e dönüştürür. Daha sonra bu değeri sayfa tablosu taban yazmacı tarafından işaret edilenPTE dizisine bir dizin olarak kullanırız.

Bu fiziksel adres bilindiğinde, donanım PTE'yi bellekten alabilir, PFN'yi çıkarabilir ve istenen fiziksel adresi oluşturmak için sanal adresten ofset ile birleştirebilir. Özellikle, PFN'nin `SHIFT` ile sola kaydırıldığını ve ardından son adresi oluşturmak için ofset ile bitsel OR'landığını aşağıdaki gibi düşünebilirsiniz:

```
offset= VirtualAddress & OFFSET_MASKPhysAddr = (PFN <<
SHIFT) | offset
```

Son olarak, donanım istenen veriyi bellekten alabilir ve `eaxyazmacına` yerleştirebilir. Program artık bellekten bir değer yüklemeyibaşarmıştır!

Özetlemek gerekirse, şimdi her bellek başvurusunda ne olacağına ilişkin ilk protokolü açıklıyoruz. Şekil 18.6 (sayfa 9) yaklaşımı göstermektedir. Her bellek başvurusu için (ister bir komut getirme isterse açık bir yükleme ya da depolama olsun), sayfalama, sayfa tablosundan çeviriyi ilk olarak getirmek için fazladan bir bellek başvurusu yapmamızı gerektirir. Bu çok fazla

```
1  // Extract the VPN from the virtual address
2  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3
4  // Form the address of the page-table entry (PTE)
5  PTEAddr = PTBR + (VPN * sizeof(PTE))
6
7  // Fetch the PTE
8  PTE = AccessMemory(PTEAddr)
9
10 // Check if process can access the page
11 if (PTE.Valid == False)
12     RaiseException(SEGMENTATION_FAULT)
13 else if (CanAccess(PTE.ProtectBits) == False)
14     RaiseException(PROTECTION_FAULT)
15 else
16     // Access is OK: form physical address and fetch it
17     offset   = VirtualAddress & OFFSET_MASK
18     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19     Register = AccessMemory(PhysAddr)
```

Şekil 18.6: **Sayfalama ile Belleğe Erişim**

çalışın! Ekstra bellek referansları maliyetlidir ve bu durumda sürecimuhtemelen iki veya daha fazla kat yavaşlatacaktır.
Ve şimdi çözmemiz gereken *iki* gerçek sorun olduğunu umarım görebilirsiniz. Hem donanım hem de yazılım dikkatli bir şekilde tasarlanmadığı takdirde, sayfa tabloları sistemin çok yavaş çalışmasına ve çok fazla bellek kullanmasına neden olacaktır. Bellek sanallaştırma ihtiyaçlarımız için harika bir çözüm gibi görünse de, öncelikle bu iki önemli sorunun üstesinden gelinmesi gerekir.

18.5 Bir Hafıza İzi

Kapatmadan önce, sayfalama kullanıldığında ortaya çıkan tüm bellek erişimlerini göstermek için şimdi basit bir bellek erişimi incelemesi yapacağız. İlgilendiğimiz kod parçacığı (C dilinde, `array.c` adlı bir dosyada) aşağıdaki gibidir:

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

We compile `array.c` and run it with the following commands:

```
prompt> gcc -o array array.c -Wall -O
prompt> ./array
```

Elbette, bu kod parçasının (sadece bir diziyi başlatan) hangi bellek erişimlerini yapacağını gerçekten anlamak için birkaç şeyi daha bilmemiz (veya varsaymamız) gerekir. İlk olarak, bir döngü içinde diziyi başlatmak için hangi assembly talimatlarının kullanıldığını görmek için sonuç ikili dosyasını (Linux'ta `objdump` veya Mac'te `otool kullanarak`) **parçalara ayırmamız gerekecek(dissasemble).** İşte ortaya çıkan assembly kodu:

```
1024 movl $0x0,(%edi,%eax,4)
1028 incl %eax
1032 cmpl $0x03e8,%eax
1036 jne  0x1024
```

Eğer biraz **x86** biliyorsanız, kodu anlamak aslında oldukça kolaydır[2]. İlk komut sıfır değerini (`$0x0` olarak gösterilir) dizinin konumunun gerçek bellek adresine taşır; bu adres `%edi`'nin içeriği alınarak ve buna dört ile çarpılmış `%eax` eklenerek hesaplanır. Böylece, `%edi` dizinin temel adresini tutarken, `%eax` dizinin indeksini (`i`) tutar; dizi her biri dört bayt boyutunda bir tamsayı dizisi olduğu için dört ile çarpıyoruz.

İkinci komut `%eax` içinde tutulan dizi indeksini artırır ve üçüncü komut bu kaydın içeriğini `0x03e8` hex değeriyle ya da 1000 ondalık değeriyle karşılaştırır. Karşılaştırma iki değerin henüz eşit olmadığını gösterirse (`jne` komutu bunu test eder), dördüncü komut döngünün başına geri atlar.

Bu komut dizisinin hangi bellek erişimlerini yaptığını anlamak için (hem sanal hem de fiziksel seviyelerde), kod parçacığının ve dizinin sanal bellekte nerede bulunduğunun yanı sıra sayfa tablosunun içeriğive konumu hakkında bir şeyler varsaymamız gerekecektir.

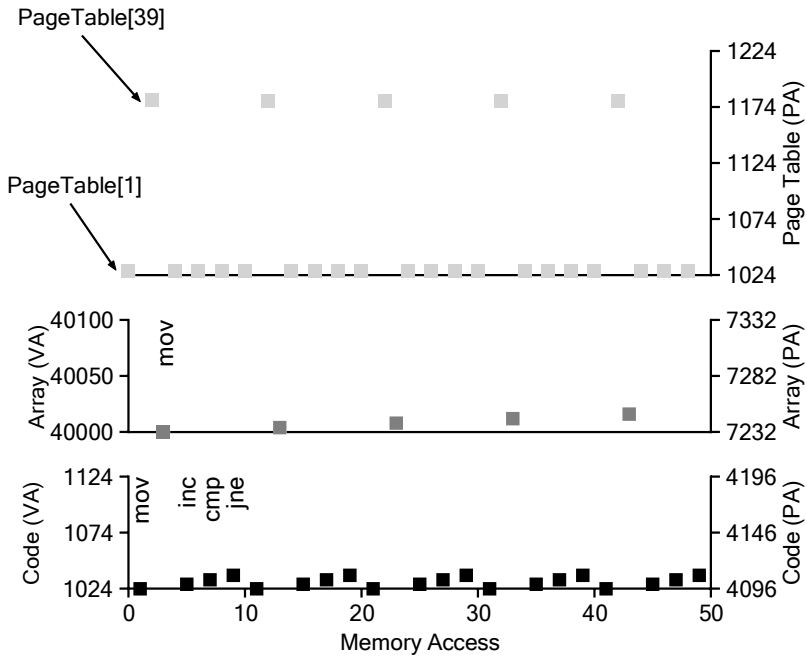
Bu örnek için 64KB boyutunda (gerçekçi olmayan bir şekilde küçük) bir sanal adres alanı varsayıyoruz. Ayrıca sayfa boyutunun 1KB olduğunu varsayıyoruz.

Şimdi bilmemiz gereken tek şey sayfa tablosunun içeriği ve fiziksel bellekteki konumudur. Doğrusal (dizi tabanlı) bir sayfa tablomuz olduğunu ve 1KB (1024) fiziksel adresinde bulunduğunu varsayalım.

İçeriğine gelince, bu örnek için eşlenmiş olması konusunda endişelenmemiz gereken sadece birkaç sanal sayfa vardır. İlk olarak, kodun üzerinde yaşadığı sanal sayfa vardır. Sayfa boyutu 1KB olduğu için, 1024 sanal adresi sanal adres alanının ikinci sayfasında bulunur (VPN=0 ilk sayfa olduğu için VPN=1). Bu sanal sayfanın fiziksel çerçeve 4 ile eşleştiğini varsayalım (VPN 1 → PFN 4).

Sonra, dizinin kendisi var. Boyutu 4000 bayttır (1000 tamsayı) ve 40000ile 44000 arasındaki sanal adreslerde (son bayt dahil değil) bulunduğunu varsayıyoruz. Bu ondalık aralık için sanal sayfalar VPN=39 ... VPN=42'dir. Dolayısıyla, bu sayfalar için eşlemelere ihtiyacımız var. Örnek için şu sanal-fiziksel eşleştirmeleri varsayalım: (VPN 39 → PFN 7), (VPN 40 → PFN 8), (VPN 41 → PFN 9), (VPN 42 → PFN 10).

[2]We are cheating a little bit here, assuming each instruction is four bytes in size for simplicity; in actuality, x86 instructions are variable-sized.



Şekil 18.7: **Bir Sanal (Ve Fiziksel) Bellek İzi**

Artık programın bellek referanslarını izlemeye hazırız. Program çalıştığında, her komut getirme işlemi iki bellek referansı oluşturacaktır:biri komutun içinde bulunduğu fiziksel çerçeveyi bulmak için sayfa tablosuna, diğeri de işlenmek üzere CPU'ya getirmek için komutun kendisine. Buna ek olarak, mov komutu şeklinde bir açık bellek referansı vardır; bu, önce başka bir sayfa tablosu erişimi (dizi sanal adresini doğru fiziksel adrese çevirmek için) ve ardından dizi erişimininkendisini ekler.

İlk beş döngü iterasyonu için tüm süreç Şekil 18.7'de (sayfa 11) gösterilmiştir. En alttaki grafik y eksenindeki komut belleği referanslarını siyah renkte göstermektedir (solda sanal adresler ve sağda gerçek fiziksel adresler); ortadaki grafik dizi erişimlerini koyu gri renkte göstermektedir (yine solda sanal ve sağda fiziksel adresler); son olarak, en üstteki grafik sayfa tablosu bellek erişimlerini açık gri renkte göstermektedir (bu örnekteki sayfa tablosu fiziksel bellekte bulunduğu için sadece fizikseldir). İzin tamamı için x ekseni, döngünün ilk beş yinelemesi boyunca bellek erişimlerini gösterir; döngü başına 10 bellek erişimi vardır, bu da dört komut getirme, bir açık bellek güncellemesi vebu dört getirme ve bir açık güncellemeyi çevirmek için beş sayfa tablosuerişimini içerir.

Bu görselleştirmede ortaya çıkan örüntüleri anlamlandırıp anlamlandıramayacağınıza bakın. Özellikle, döngü bu ilk beş iterasyonun ötesinde çalışmaya devam ettikçe ne değişecek? Hangi yeni bellek konumlarına erişilecek? Bunu çözebilir misiniz?

Bu sadece en basit örnekti (sadece birkaç satır C kodu), ancak yine de gerçek uygulamaların gerçek bellek davranışını anlamanın karmaşıklığını şimdiden hissedebilirsiniz. Endişelenmeyin: durum kesinlikle daha da kötüleşecek, çünkü birazdan tanıtacağımız mekanizmalar zaten karmaşık olan bu mekanizmayı daha da karmaşıklaştıracak. Üzgünüm[3] !

!

18.6 Özet

Belleği sanallaştırma sorunumuza bir çözüm olarak **sayfalama** kavramını tanıttık. Sayfalama, önceki yaklaşımlara (segmentasyon gibi) göre birçok avantaja sahiptir. İlk olarak, sayfalama (tasarım gereği) belleği sabit boyutlu birimlere böldüğü için harici parçalanmaya yol açmaz. İkinci olarak, oldukça esnektir ve sanal ad- dress alanlarının seyrek kullanımına olanak tanır.

Ancak, disk belleği desteğinin dikkatli bir şekilde uygulanmaması makinenin yavaşlamasına (sayfa tablosuna erişmek için çok sayıda ekstra bellek erişimi) ve bellek israfına (belleğin yararlı uygulama verileri yerine sayfa tablolarıyla dolmasına) yol açacaktır. Bu nedenle, sadece çalışan değil, aynı zamanda iyi çalışan bir disk belleği sistemi bulmak için biraz daha fazla düşünmemiz gerekecektir. Neyse ki önümüzdeki iki bölüm bize bunu nasıl yapacağımızı gösterecek.

[3]We're not really sorry. But, we are sorry about not being sorry, if that makes sense.

References

[KE+62] "One-level Storage System" by T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sumner. IRE Trans. EC-11, 2, 1962. Reprinted in Bell and Newell, "Computer Structures: Readings and Examples". McGraw-Hill, New York, 1971. *The Atlas pioneered the idea of dividing memory into fixed-sized pages and in many senses was an early form of the memory-management ideas we see in modern computer systems.*

[I09] "Intel 64 and IA-32 Architectures Software Developer's Manuals" Intel, 2009. Available: http://www.intel.com/products/processor/manuals. *In particular, pay attention to "Volume 3A: System Programming Guide Part 1" and "Volume 3B: System Programming Guide Part 2".*

[L78] "The Manchester Mark I and Atlas: A Historical Perspective" by S. H. Lavington. Communications of the ACM, Volume 21:1, January 1978. *This paper is a great retrospective of some of the history of the development of some important computer systems. As we sometimes forget in the US, many of these new ideas came from overseas.*

Ev Ödevi (Simülasyon)

Bu ödevde, `paging-linear-translate.py` olarak bilinen basit bir programı kullanarak, basit sanal-fiziksel adres çevirisinin doğrusal sayfa tablolarıyla nasıl çalıştığını anlayıp anlamadığınızı göreceksiniz. Ayrıntılar için README'ye bakın.

Sorular

1. Herhangi bir çeviri yapmadan önce, farklı parametreler verildiğinde doğrusal sayfa tablolarının boyutunun nasıl değiştiğini incelemek için simülatörü kullanalım. Farklı parametreler değiştikçe doğrusal sayfa tablolarının boyutunu hesaplayın. Önerilen bazı girdiler aşağıdadır; `-v` bayrağını kullanarak kaç sayfa tablosu girdisinin doldurulduğunu görebilirsiniz. İlk olarak, adres alanı büyüdükçe doğrusal sayfa tablosu boyutunun nasıl değiştiğini anlamak için bu bayraklarla çalıştırın:

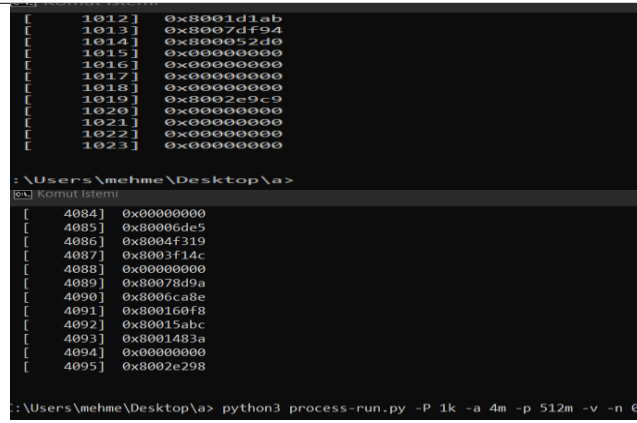
```
-P 1k -a 1m -p 512m -v -n 0
-P 1k -a 2m -p 512m -v -n 0
-P 1k -a 4m -p 512m -v -n 0
```

Ardından, sayfa boyutu büyüdükçe doğrusal sayfa tablosu boyutunun nasıl değiştiğini anlamak için:

```
-P 1k -a 1m -p 512m -v -n 0
-P 2k -a 1m -p 512m -v -n 0
-P 4k -a 1m -p 512m -v -n 0
```

Bunlardan herhangi birini çalıştırmadan önce, beklenen eğilimler hakkında düşünmeye çalışın. Adres alanı büyüdükçe sayfa tablosu boyutu nasıl değişmelidir? Sayfa boyutu büyüdükçe? Neden genel olarak büyük sayfalar kullanılmıyor?

Adres alanı(ADRESS SPACE GROWS) büyüdükçe Doğrusal Sayfa tablosu boyutu(LINEAR PAGE- TABLE SIZE) da aynı miktarda büyümektedir. Fakat sayfa boyutu büyüdükçe doğrusal sayfa boyutu azalma göstermektedir. İlk çalıştırıldığında aşağıda ki görselde göründüğü gibi 4 satır olan sayfa tablosu boyutu, adres alanı büyütüldükçe aynı oranda artış göstermektedir.



Büyük sayfalar kullanılmasının sebebi ise sayfa boyutu büyüdükçe bellek yönetiminin daha zor hale getirmesi ve bunun da işletim sisteminin performansını düşürmesinden dolayıdır.

2. Şimdi bazı çeviriler yapalım. Bazı küçük örneklerle başlayın ve `-u` bayrağı ile adres alanına ayrılan sayfa sayısını değiştirin. Örneğin:

```
-P 1k -a 16k -p 32k -v -u 0
-P 1k -a 16k -p 32k -v -u 25
-P 1k -a 16k -p 32k -v -u 50
-P 1k -a 16k -p 32k -v -u 75
-P 1k -a 16k -p 32k -v -u 100
```

Her bir adres alanında bulunan sayfaların yüzdesini artırdığınızda ne olur?

Belirtilen bellek boyutlarına göre sayfalama (PAGING) işlemi yapan bu komutların değerleri düzgün bir şekilde belirtilmiş ise paging işlemi sırasında sayfaların yüzdesinin arttırmak sistemin performansını ve verimliliğimi arttırabilir. Sistemin bellek kullanımını da optimize etmeye yardımcı olabilir. Ancak bu işlemi yaparken bellek boyutlarının ve adres alanında bulunan yüzde doğru şekilde ayarlanması gereklidir

3. Şimdi çeşitlilik için bazı farklı rastgele tohumlar ve bazı farklı (ve bazen oldukça çılgın) adres alanı parametrelerini deneyelim:

```
-P 8   -a 32    -p 1024 -v -s 1
-P 8k  -a 32k   -p 1m   -v -s 2
-P 1m  -a 256m  -p 512m -v -s 3
```

Bu parametre kombinasyonlarından hangileri gerçekçi değildir? Neden?

Bu parametre kombinasyonlarından 1 ve 2. Kombinasyon gerçekçi değildir. Nedeni bellek boyutlarının belirtlimesinde ki hatalardır. -P ve -a parametreleri kilobyte birimini kullanırken -p parametresi ise megabyte birimini kullanır. Bu sebeple yanlış verilen parametrelerden ötürü çalışmamaktadır. Aşağıda ki görsellerde ise her parametrenin invalid değerlerden ötürü kullanılamadığı gösterilmiştir.

-P 8 -a 32 -p 1024 -v -s 1

```
C:\Users\mehme> cd desktop/a

C:\Users\mehme\Desktop\a> python3 process-run.py -P 8 -a 32-p 1024 -v -s 1
ARG seed 1
ARG address space size 32-p
ARG phys mem size 64k
ARG page size 8
ARG verbose True
ARG addresses -1

Traceback (most recent call last):
  File "C:\Users\mehme\Desktop\a\process-run.py", line 63, in <module>
    asize    = convert(options.asize)
  File "C:\Users\mehme\Desktop\a\process-run.py", line 30, in convert
    nsize = int(size)
ValueError: invalid literal for int() with base 10: '32-p'
```

-P 8k -a 32k -p 1m   -v -s 2

```
C:\Users\mehme\Desktop\a> python3 process-run.py -P 8k -a 32k -p 1m-v -s 2
ARG seed 2
ARG address space size 32k
ARG phys mem size 1m-v
ARG page size 8k
ARG verbose False
ARG addresses -1

Traceback (most recent call last):
  File "C:\Users\mehme\Desktop\a\process-run.py", line 64, in <module>
    psize    = convert(options.psize)
  File "C:\Users\mehme\Desktop\a\process-run.py", line 30, in convert
    nsize = int(size)
ValueError: invalid literal for int() with base 10: '1m-v'
```

4. Diğer bazı problemleri denemek için programı kullanın. Programın artık çalışmadığı sınırları bulabilir misiniz? Örneğin, adres alanı boyutu fiziksel bellekten daha *büyükse* ne olur?

Adres alanı boyutu(ADRESS SPACE GROWS) fiziksel adresten (PHYSICAL ADDRESS) büyük olursa, bellek yönetimi yapılırken hata

mesajı alınabilir. Sistemde bellek alanının yetersiz olduğunu gösterir ve sistem çalışmaz. Adres alanı boyutunun fiziksel adres boyutuna eşit veya daha küçük olması gereklidir.