18

Paging: Introduction

(Sayfalama: Giriş)

Bazen işletim sisteminin herhangi bir alan yönetimi probleminin çoğunu çözerken iki yaklaşımdan birini aldığı söylenir. İlk yaklaşım, sanal bellekte **segmentasyonda(segmentation)** gördüğümüz gibi, sanal belleği *değişken boyutlu* parçalara ayırırız. Ne yazık ki, bu çözümün doğal zorlukları vardır. Özellikle, bir belleği farklı büyüklükteki parçalara bölerken, belleğin kendisi **parçalanabilir(fragmented**) ve böylece alan tahsisi zamanla daha zor hale gelir.

Bu nedenle, ikinci yaklaşımı göz önünde bulundurmaya değebilir: alanı *sabit boyutlu* parçalara ayırmak. Sanal bellekte, bu fikre **sayfalama(paging)** diyoruz ve erken ve önemli bir sistem olan Atlas'a [KE + 62, L78] geri dönüyor. Bir işlemin adres alanını bir dizi değişken boyutlu mantıksal segmente (örneğin, kod, yığın) bölmek yerine, her birine **sayfa(page)** dediğimiz sabit boyutlu birimlere böleriz. Buna paralel olarak, fiziksel belleği sayfa çerçeveleri adı verilen sabit boyutlu yuvalar dizisi olarak görüyoruz; bu **page frames**'lerin her biri tek bir sanal bellek sayfası içerebilir. Karşılaştığımız zorluk:

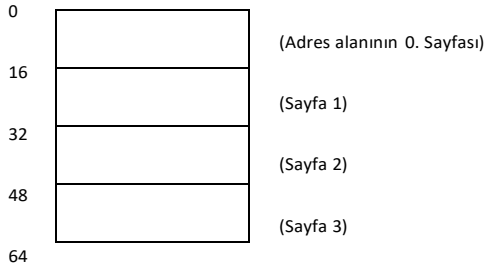
ÖNEMLİ NOKTA:
SAYFALARLA BELLEĞI SANALLAŞTIRMA

Segmentasyon sorunlarından kaçınmak için belleği sayfalarla nasıl sanallaştırabiliriz? Temel teknikler nelerdir? Bu tekniklerin asgari alan ve zaman yükü ile iyi çalışmasını nasıl sağlayabiliriz?

18.1 Basit Bir Örnek ve Genel Bakış

Bu yaklaşımı daha açık hale getirmeye yardımcı olmak için, basit bir örnekle gösterelim. Şekil 18.1 (sayfa 2), dört adet 16 baytlık sayfa (sanal sayfalar 0, 1, 2 ve 3) ile toplam 64 bayt boyutunda küçük bir adres alanı örneği sunmaktadır. Gerçek adres alanları çok daha büyüktür, elbette, genellikle 32 bit ve böylece 4 GB adres alanı, hatta 64 bit[1]; Kitapta, sindirimlerini kolaylaştırmak için genellikle küçük örnekler kullanacağız.

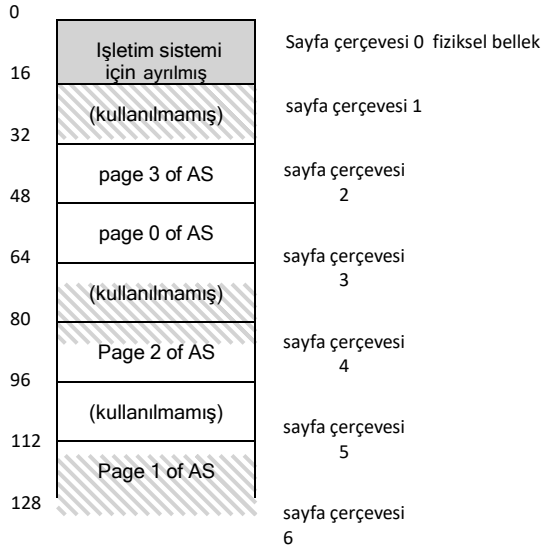
[1]64 bitlik bir adres alanını hayal etmek zor, inanılmaz derecede büyük. Bir benzetme yardımcı olabilir: 32 bitlik bir adres alanını tenis kortunun büyüklüğü olarak düşünürseniz, 64 bitlik bir adres alanı Avrupa'nın (!) büyüklüğü kadardır.



Şekil 18.1: **Basit bir 64 baytlık adres alanı**

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

Sayfalama, göreceğimiz gibi, önceki yaklaşımlarımıza göre birtakım avantajlara sahiptir. Muhtemelen en önemli gelişme *esneklik* olacaktır: tamamen gelişmiş bir sayfalama yaklaşımıyla, sistem, bir işlemin adres alanını nasıl kullandığına bakılmaksızın, bir adres alanının bölümlendirilmesi etkili bir şekilde destekleyebilecektir; örneğin, varsayımda bulunmayacağız. Yığın ve yığının büyüme yönü ve nasıl kullanıldıkları hakkında bilgiler.



Şekil 18.2: A 64-Byte Address Space In A 128-Byte Physical Memory

Diğer bir avantaj, sayfalamanın sağladığı boş alan yönetiminin *basitliğidir*. Örneğin, işletim sistemi küçük 64 baytlık adres alanımızı sekiz sayfalık fiziksel belleğimize yerleştirmek istediğinde, sadece dört boş sayfa bulur; belki de işletim sistemi bunun için tüm **free list**'ler için boş bir liste tutar ve sadece ilk dört boş sayfayı bu listeden alır. Örnekte, işletim sistemi adres alanının (AS) sanal sayfa 0'ını fiziksel çerçeve 3'e, AS'nin sanal sayfa 1'ini 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 ücretsizdir.

Adres alanının her sanal sayfasının fiziksel bellekte nereye yerleştirildiğini kaydetmek için, işletim sistemi genellikle **sayfa tablosu (page table)** olarak bilinen *işlem başına* bir veri yapısı tutar. Sayfa tablosunun ana rolü, adres alanının sanal sayfalarının her biri için **adres çevirilerini (adress translations)** depolamak ve böylece her sayfanın fiziksel bellekte nerede bulunduğunu bize bildirmektir. Basit örneğimiz için (Şekil 18.2, sayfa 2), sayfa tablosu aşağıdaki dört girişe sahip olacaktır: (Sanal Sayfa 0 → Fiziksel Çerçeve 3), (VP 1 → PF 7), (VP 2 → PF 5), ve (VP 3 → PF 2).

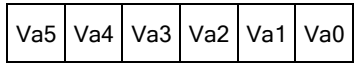
Bu sayfa tablosunun işlem başına bir veri yapısı olduğunu hatırlamak önemlidir (tartıştığımız çoğu sayfa tablosu yapısı işlem başına yapılardır; değineceğimiz bir istisna **ters çevrilmiş sayfa tablosudur (inverted page table).** Yukarıdaki örneğimizde başka bir işlem çalışacak olsaydı, işletim sisteminin bunun için farklı bir sayfa tablosu yönetmesi gerekirdi, çünkü v irtual sayfaları açıkça *farklı* fiziksel sayfalarla eşleşir (modül herhangi bir paylaşıma devam eder).

Şimdi, bir adres çevirisi örneği gerçekleştirmek için yeterli bilgiye sahibiz. Bu küçük adres alanına (64 bayt) sahip işlemin bir bellek erişimi oluşturduğunu hayal edelim:

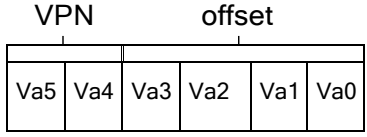
```
Movl <sanal adres>, %eax
```

Özellikle, verilerin adres <sanal adres> kayıt eax içine açık yüküne dikkat edelim (ve daha önce gerçekleşmesi gereken talimat getirmeyi görmezden gelin).

İşlemin oluşturduğu bu sanal adresi **çevirmek(translate)** için önce iki bileşene bölmemiz gerekir: **sanal sayfa numarası (VPN**) ve sayfa içindeki **offset**. Bu örnek için işlemin sanal adres alanı 64 bayt olduğundan, sanal adresimiz için toplam 6 bit'e ihtiyacımız var.
(2^6 = 64). Böylece sanal adresimiz şu şekilde kavramsallaştırılabilir:



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

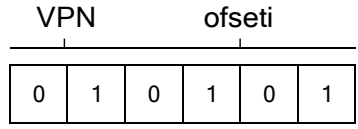


Sayfa boyutu 64 baytlık bir adres alanında 16 bayttır; bu nedenle 4 sayfa seçebilmemiz gerekir ve adresin en üstteki 2 biti tam da bunu yapar. Böylece, 2 bitlik bir sanal sayfa numarasına (VPN) sahibiz. Kalan bitler bize sayfanın hangi baytıyla ilgilendiğimizi söyler, bu durumda 4 bit; buna ofset diyoruz.

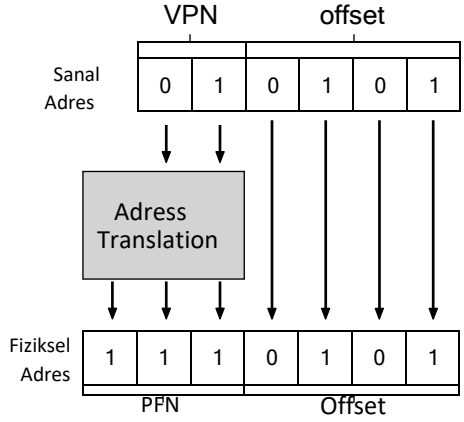
Bir işlem sanal bir adres oluşturduğunda, işletim sistemi ve donanımın anlamlı bir fiziksel adrese çevirmek için birleştirilmesi gerekir. Örneğin, yukarıdaki yükün sanal adres 21'e ait olduğunu varsayalım:

```
movl 21, %eax
```

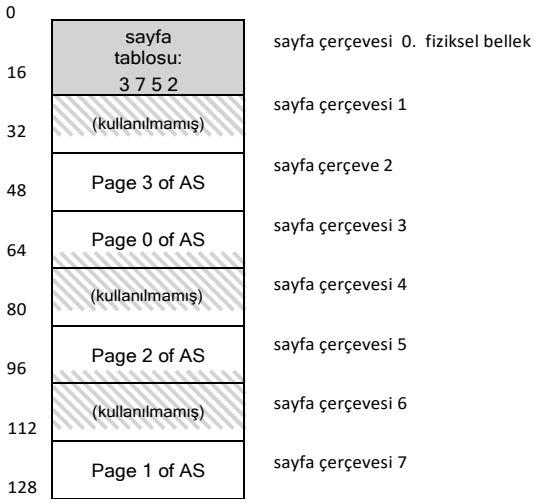
"21" i ikili forma dönüştürerek , "010101" alırız ve böylece bu sanal adresi deneyebilir ve sanal bir sayfa numarasına (VPN) nasıl ayrıldığını görebilir ve ofsetlenebilir:



Bu nedenle, "21" sanal adresi, "01" (veya 1) sanal sayfasının 5. ("0101"th) baytındadır. Sanal sayfa numaramızla, artık sayfa tablomuzu dizine ekleyebilir ve sanal sayfa 1'in hangi fiziksel çerçevede bulunduğunu bulabiliriz. Yukarıdaki sayfa tablosunda **fiziksel çerçeve numarası (PFN)** (bazen **fiziksel sayfa numarası** veya **PPN** olarak da adlandırılır) 7'dir (binary 111). Böylece, VPN'i PFN ile değiştirerek bu sanal adresi çevirebilir ve ardından yükü fiziksel belleğe verebiliriz (Şekil 18.3).



Şekil 18.3: **Adres Çeviri İşlemi (Adress Translation Process)**



Şekil 18.4: **Örnek:** Page Table in Kernel Physical Memory

Uzaklık aynı kalır (yani, çevrilmez), çünkü uzaklık bize yalnızca sayfa *içinde* hangi baytı istediğimizi söyler. Son fiziksel adresimiz 1110101 (ondalık olarak 117) ve tam olarak yükümüzün veri almasını istediğimiz yerdir (Şekil 18.2, sayfa 2).

Bu temel genel bakışı göz önünde bulundurarak, şimdi sayfalama hakkında sahip olabileceğiniz birkaç temel soruyu sorabiliriz (ve umarım cevaplayabiliriz). Örneğin, bu sayfa tabloları nerede depolanır? Sayfa tablosunun tipik içeriği nelerdir ve tablolar ne kadar büyüktür? Disk belleği sistemi (de) yavaşlatır mı? Bu ve diğer aldatıcı sorular, en azından kısmen, aşağıdaki metinde cevaplandırılmıştır . Okumaya devam edin!

18.2 Sayfa Tabloları Nerede Depolanır?

Sayfa tabloları , daha önce tartıştığımız küçük segment tablosundan veya taban / sınır çiftinden çok daha büyük olabilir. Örneğin, 4 KB sayfalı tipik bir 32 bit adres alanı düşünün. Bu sanal reklam elbisesi 20 bit VPN ve 12 bit ofsete ayrılır (1 KB'lık bir sayfa boyutu için 10 bit'in gerekli olacağını unutmayın ve 4 KB'a ulaşmak için iki bit daha ekleyin).

20 bitlik bir VPN, işletim sisteminin her işlem için yönetmesi gereken 2^{20} çeviri olduğu anlamına gelir (bu kabaca bir milyondur); Fiziksel çeviriyi ve diğer yararlı şeyleri tutmak için sayfa başına 4 bayt **tablo girişine (PTE)** ihtiyacımız olduğunu varsayarsak, her sayfa tablosu için gereken muazzam bir 4MB bellek elde ederiz! Bu oldukça büyük. Şimdi çalışan 100 işlem olduğunu hayal edin: bu, işletim sisteminin yalnızca tüm bu adres çevirileri için 400MB belleğe ihtiyaç duyacağı anlamına gelir! Modern çağda bile, nerede

ASIDE: DATA STRUCTURE — THE PAGE TABLE

Modern bir işletim sisteminin bellek yönetimi alt sistemindeki en önemli veri yapılarından biri **sayfa tablosudur (Page table)**. Genel olarak, bir sayfa tablosu **sanaldan fiziksele adres çevirilerini saklar (virtual-to-physical adress translation)**, böylece sisteme bir adres alanının her sayfasının gerçekte fiziksel bellekte nerede bulunduğunu bildirir. Her adres alanı bu tür çevirileri gerektirdiğinden, genel olarak sistemdeki işlem başına bir sayfa tablosu vardır. Sayfa tablosunun tam yapısı donanım (eski sistemler) tarafından belirlenir veya işletim sistemi (modern sistemler) tarafından daha esnek bir şekilde yönetilebilir.

makinelerin gigabaytlarca belleği var, büyük bir kısmını sadece çeviriler için kullanmak biraz çılgınca görünüyor , değil mi? Ve böyle bir sayfa tablosunun 64 bitlik bir adres alanı için ne kadar büyük olacağını düşünmeyeceğiz bile; bu çok korkunç olurdu ve belki de sizi tamamen korkuturdu.

Sayfa tabloları çok büyük olduğundan, şu anda çalışan işlemin sayfa tablosunu depolamak için MMU'da herhangi bir özel yonga üstü donanım tutmuyoruz. Bunun yerine, her işlem için sayfa tablosunu bellekte bir yerde saklarız . Şimdilik, sayfa tablolarının işletim sisteminin yönettiği fiziksel bellekte yaşadığını varsayalım; Daha sonra işletim sistemi belleğinin muc h'sinin virtüalize edilebileceğini ve böylece sayfa tablolarının işletim sistemi sanal belleğinde saklanabileceğini (ve hatta diske değiştirilebileceğini) göreceğiz, ancak bu şu anda çok kafa karıştırıcı, bu yüzden bunu yapacağız. Şekil 18.4'te (sayfa 5), işletim sistemi belleğindeki bir sayfa tablosunun resmidir; oradaki küçük çeviri setini görüyor musunuz?

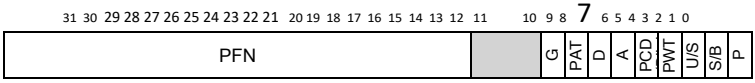
18.3 Sayfa Tablosunda Aslında Neler Var?

Sayfa tablosu organizasyonu hakkında biraz konuşalım. Sayfa tablosu

yalnızca sanal adresleri (veya gerçekten sanal sayfa numaralarını) fiziksel adreslere (fiziksel çerçeve numaraları) eşlemek için kullanılan bir veri yapısıdır. Böylece, herhangi bir veri yapısı çalışabilir. En basit forma, yalnızca bir dizi olan **doğrusal sayfa tablosu (linear page table)** denir. İşletim sistemi, diziyi sanal sayfa numarasına (VPN) göre *dizine ekler* ve istenen fiziksel çerçeveyi bulmak için bu dizindeki sayfa tablosu girişini (PTE) arar. numarası (PFN). Şimdilik, bu basit doğrusal yapıyı varsayacağız; Daha sonraki bölümlerde, sayfalama ile ilgili bazı sorunların çözülmesine yardımcı olmak için daha gelişmiş veri yapılarından yararlanacağız.

Her PTE'nin içeriğine gelince, orada bir seviyede anlaşılmaya değer bir dizi farklı bitimiz var. Belirli bir çevirinin geçerli olup olmadığını belirtmek için **geçerli bir bit (a valid bit)** yaygındır; örneğin, bir program çalışmaya başladığında, adres alanının bir ucunda e kodu ve yığını, diğer ucunda ise yığın olacaktır. Aradaki tüm kullanılmayan boşluklar **geçersiz(invalid)** olarak işaretlenir ve işlem bu belleğe erişmeye çalışırsa, işletim sistemine muhtemelen işlemi sonlandıracak bir tuzak oluşturur. Bu nedenle, geçerli bit seyrek bir adres alanını desteklemek için çok

önemlidir; Adres alanındaki kullanılmayan tüm sayfaları geçersiz olarak işaretleyerek, bu sayfalar için fiziksel çerçeveler ayırma ihtiyacını ortadan kaldırırız ve böylece büyük miktarda bellek tasarrufu sağlarız.



Şekil 18.5: **x86 Sayfa Tablo Girişi (PTE)**

Ayrıca, sayfanın okunup okunamayacağını, yazılıp yazılamayacağını veya yürütülüp yürütülemeyeceğini belirten **koruma bitlerimiz (protection bits)** de olabilir. Yine, bir sayfaya bu bitlerin izin vermediği bir şekilde erişmek, işletim sistemine bir tuzak oluşturacaktır.

Önemli olan birkaç parça daha var, ancak şimdilik fazla konuşmayacağız. Mevcut bir **bit**, bu sayfanın fiziksel bellekte mi yoksa diskte mi olduğunu gösterir (yani, değiştirilmiştir **(swapped out)**). Fiziksel bellekten daha büyük adres alanlarını desteklemek için adres alanının bölümlerinin diske nasıl **değiştirileceğini(swap)** incelediğimizde bu makineyi daha da destekleyeceğiz; değiştirmek, işletim sisteminin nadiren kullanılan sayfaları diske taşıyarak fiziksel belleği boşaltmasına izin verir. Sayfanın belleğe alınmasından bu yana değiştirilip değiştirilmediğini gösteren **kirli bir bit (dirty bit)** de yaygındır.

Bir **referans biti** (diğer adıyla **accessed bit**) bazen kullanılıp kullanılmayacağını izlemek için kullanılır**.**

Bir sayfaya erişilmiştir ve hangi sayfaların popüler olduğunu ve bu nedenle bellekte tutulması gerektiğini belirlemede yararlıdır; bu tür bilgiler, **sayfa değiştirme (page replacement)** sırasında kritik öneme sahiptir, sonraki bölümlerde ayrıntılı olarak inceleyeceğimiz bir konudur.

Şekil 18.5, x86 armatüründen [I09] örnek bir sayfa tablosu girişini göstermektedir. Mevcut bir bit (P) içerir; bu sayfaya yazmaya izin verilip verilmeyeceğini belirleyen bir okuma / yazma biti (R / W); kullanıcı modu işlemlerinin sayfaya erişip erişemeyeceğini belirleyen bir kullanıcı/gözetmen biti (U/S); donanım önbelleğe almanın bu sayfalar için nasıl çalıştığını belirleyen birkaç bit (PWT, PCD, PAT ve G); erişilen bir bit (A) ve kirli bir bit (D); ve son olarak, sayfa çerçeve numarasının (PFN) kendisi.

x86 sayfalama desteği hakkında daha fazla bilgi için Intel mimarisi kılavuzlarını [I09] okuyun. Bununla birlikte, önceden uyarılmalıdır; kılavuzları bu şekilde okumak, oldukça bilgilendirici olsa da (ve işletim sisteminde bu tür sayfa tablolarını kullanmak için kod yazanlar için kesinlikle gerekli), ilk başta zor olabilir. Biraz eğlence ve çok fazla arzu gereklidir.

Ayrıca: NEDEN GEÇERLI BIT YOK?

Intel örneğinde, ayrı geçerli ve mevcut bitler olmadığını, bunun yerine sadece mevcut bir bit (P) olduğunu fark edebilirsiniz. Bu bit ayarlanmışsa (P = 1), sayfanın hem mevcut hem de geçerli olduğu anlamına gelir. Değilse (P = 0), sayfanın bende mory bulunmayabileceği (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 tekrar değiştirilmesi gerektiğini) belirlemek için tuttuğu ek yapıları kullanmalıdır (ve bu nedenleprogram belleğe yasadışı olarak erişmeye çalışmaktadır). Bu tür bir mantıklılık, genellikle işletim sisteminin tam bir hizmet oluşturabileceği minimum özellik kümesini sağlayan donanımda yaygındır.

18.4 Disk Belleği: Ayrıca Çok Yavaş

Bellekteki sayfa tablolarıyla , çok büyük olabileceklerini zaten biliyoruz. Görünüşe göre, işleri de yavaşlatabilirler. Örneğin, basit talimatımızı ele alalım:

```
movl 21, %eax
```

Yine, adres 21'e yapılan açık referansı inceleyelim ve talimat getirme konusunda endişelenmeyelim. Bu örnekte, donanım yazılımının çeviriyi**(translate)** bizim için gerçekleştirdiğini varsayacağız. İstenilen verileri almak için, sistem önce sanal bir elbiseyi (21) doğru fiziksel reklam elbisesine (117) çevirmelidir. Bu nedenle, verileri adres 117'den almadan önce, sistemin önce işlemin sayfa tablosundan uygun sayfa tablosu girişini getirmesi, çeviriyi gerçekleştirmesi ve ardından verileri fiziksel bellekten yüklemesi gerekir .

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

```
VPN= (Sanal Adres ve VPN_MASK)>> SHIFT PTEAddr = PageTableBaseRegister
+ (VPN * sizeof (PTE))
```

Örneğimizde, VPN MASKESİ, VPN bitlerini tam sanal adresten seçen 0x30 (onaltılık 30 veya ikili 110000) olarak ayarlanır; SHIFT, doğru tamsayı sanal sayfa numarasını oluşturmak için VPN bitlerini aşağı doğru hareket ettirecek şekilde 4 (ofsetteki bit sayısı) olarak ayarlanır. İnceleme için, sanal adres 21 (010101) ile ve maskeleme bu değeri 010000'e dönüştürür; kaydırma istenildiği gibi 01'e veya sanal sayfa 1'e dönüştürür. Daha sonra bu değeri, sayfa tablosu temel kaydı tarafından işaret edilen PTE'lerin arr ay'ına bir dizin olarak kullanırız.

Bu fiziksel adres bilindikten sonra, donanım PTE'yi bellekten alabilir, PFN'yi ayıklayabilir ve istenen fiziksel adresi oluşturmak için sanal adresten ofset ile birleştirebilir . Özellikle, PFN'nin SHIFT tarafından sola kaydırıldığını ve ardından son adresi aşağıdaki gibi oluşturmak için ofsetle bitsel olarak OR'd olduğunu düşünebilirsiniz :

```
offset= Sanal Adres & OFFSET_MASK PhysAddr = (PFN
<<SHIFT) | ofset
```

Son olarak, donanım istenen verileri bellekten alabilir ve kayıt eax'ına koyabilir. Program şimdi bellekten bir değer yüklemeyi başardı!

Özetlemek gerekirse, şimdi her bellek başvurusunda ne olduğuna dair ilk protokolü açıklıyoruz. Şekil 18.6 (sayfa 9) yaklaşımı göstermektedir. Her bellek başvurusu için (talimat getirme veya açık yükleme veya depolama), disk belleği, çeviriyi sayfa tablosundan ilk olarak getirmek için fazladan bir bellek başvurusu gerçekleştirmemizi gerektirir . Bu çok fazla

```
1  // VPN'yi sanal adresten çıkarın
2  VPN = (Sanal Adres ve VPN_MASK)>> SHIFT
3
4  // Sayfa tablosu girişinin adresini oluşturun (PTE)
5  PTEAddr = PTBR + (VPN * sizeofPTE))
6
7  // PTE'yi Getir
8  PTE = AccessMemory (PTEAddr)
9
10 // İşlemin sayfaya erişip erişemediğini kontrol edin
11 if (PTE. Geçerli == Yanlış)
12       RaiseException (SEGMENTATION_FAULT)
13 else if (CanAccess (PTE. ProtectBits) == False)
14       RaiseException (PROTECTION_FAULT)
15 else
16       // Erişim Tamam: fiziksel adres oluşturun ve alın
17       ofset          = Sanal Adres ve OFFSET_MASK
18       PhysAddr = (PTE. PFN <<PFN_SHIFT) |  ofset
19       Kayıt = AccessMemory (PhysAddr)
```

Şekil 18.6: **Disk Belleği ile Belleğe Erişme**

iş! Ekstra bellek başvuruları maliyetlidir ve bu durumda işlemi iki veya daha fazla kat yavaşlatacaktır.

Ve şimdi umarım çözmemiz gereken *iki* gerçek sorun olduğunu görebilirsiniz. Hem donanım hem de yazılımın dikkatli bir şekilde tasarlanmaması durumunda, sayfa tabloları sistemin çok yavaş çalışmasına ve çok fazla bellek kaplamasına neden olur. Bellek sanallaştırma ihtiyaçlarımız için harika bir çözüm gibi görünse de bu iki önemli sorunun üstesinden gelinmelidir.

18.5 Bir Bellek İzi

Kapatmadan önce, disk belleği kullanılırken ortaya çıkan tüm bellek erişimlerini göstermek için basit bir bellek erişim incelemesinden geçiyoruz. İlgilendiğimiz kod parçacığı (C'de , array.c adlı bir dosyada) aşağıdaki gibidir:

```
Int dizisi [1000];
...
İçin (i = 0; i 1000 <; i++) dizisi[i] = 0;
```

Array.c'yi derleyip aşağıdaki komutlarla çalıştırıyoruz:

```
Istemi> gcc -o array array.c  -Wall -O istemi>. /dizi
```

Tabii ki, bu kod snip-pet'in (sadece bir diziyi başlatan) hangi belleğe eriştiğini gerçekten anlamak için, birkaç şey daha bilmemiz (veya varsaymamız) gerekecek. İlk olarak, bir döngüde diziyi başlatmak için hangi montaj talimatlarının kullanıldığını görmek için sonuç veren ikili dosyayı (Linux'ta objdump veya Mac'te otool kullanarak) **sökmemiz(disassemble)** gerekecek. İşte ortaya çıkan derleme kodu:

```
1024  harek  $0x0, (%edi, %eax,4)
         et
1028  Dahil  %eax
1032  cmpl   $0x03e8, %eax
1036  ces    0x1024
        aret
```

Kod, biraz **x86** biliyorsanız, aslında 2'yi anlamak oldukça kolaydır. İlk talimat, sıfır değerini ($0x0 olarak gösterilir) dizinin konumunun virtüal bellek adresine taşır; bu adres, %edi ve ekleme %eax içeriğinin dört ile çarpılmasıyla hesaplanır. Böylece, %edi dizinin temel adresini tutarken, %eax dizi dizinini (i) tutar; dörde çarparız, çünkü dizi , her biri dört bayt büyüklüğünde bir tamsayılar dizisidir .

İkinci yönerge, %eax'ta tutulan dizi dizinini artırır ve üçüncü yönerge, bu kaydın içeriğini 8 veya ondalık 1000'0x03e onaltılık değeriyle karşılaştırır . Karşılaştırma iki değerin henüz eşit olmadığını gösteriyorsa (jne talimatının test ettiği şey budur), dördüncü talimat döngünün en üstüne geri atlar.

Bu komut dizisinin hangi belleğe eriştiğini anlamak için (hem sanal hem de fiziksel düzeylerde), co de snippet ve dizinin sanal bellekte neredebulunduğunun yanı sıra sayfa tablosunun içeriği ve konumu hakkında bir şeyler varsaymamız gerekir.

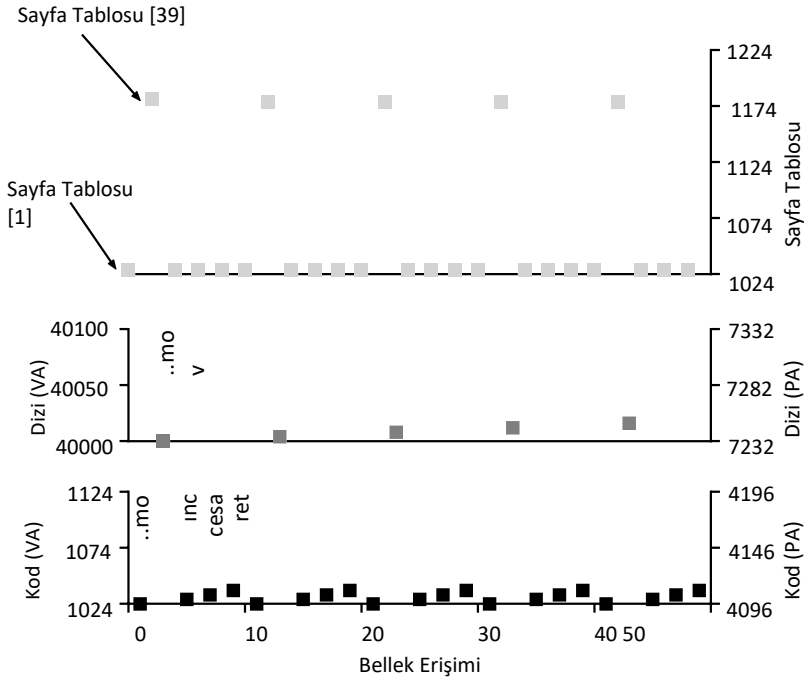
Bu örnekte, 64KB boyutunda (gerçekçi olmayan bir şekilde küçük) bir sanal adres alanı varsayıyoruz. Ayrıca 1 KB sayfa boyutu olduğunu varsayıyoruz.

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

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

Sonra, dizinin kendisi var. Boyutu 4000 bayttır (1000 tam sayı) ve 40000 ile 44000 arasındaki sanal adreslerde (son bayt hariç) kenarlarda olduğunu varsayıyoruz. Bu ondalık aralık için sanal sayfalar VPN = 39 ... VPN=42. Bu nedenle, bu sayfalar için eşlemelere ihtiyacımız var. Bu sanal-fiziksel eşlemeleri örnek olarak kabul edelim : (VPN 39 → PFN 7), (VPN 40 → PFN 8), (VPN 41 → PFN 9), (VPN 42 → PFN 10).

[2]Burada biraz hile yapıyoruz, her talimatın basitlik için dört bayt boyutunda olduğunu varsayıyoruz; gerçekte, x86 talimatları değişken boyutludur.



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

Artık programın bellek referanslarını izlemeye hazırız. Çalıştığında, her yönerge getirme işlemi iki bellek başvurusu oluşturur: biri yönergenin içinde bulunduğu fiziksel çerçeveyi bulmak için sayfa tablosuna, diğeri de yönergenin kendisine getirmek için yönergenin kendisine işleme için CPU. Ek olarak, mov talimatı biçiminde açık bir bellek başvurusu vardır; bu, önce başka bir sayfa tablosu erişimi (dizi sanal adresini doğru fiziksel adrese çevirmek için) ve ardından dizi erişiminin kendisini ekler.

İlk beş döngü yinelemesi için tüm süreç Şekil 18.7'de (sayfa 11) tasvir edilmiştir. En alttaki grafik, y eksenindeki talimat belleği referanslarını siyah renkte gösterir (solda sanal adresler ve sağda gerçek fiziksel adreslerle); ortadaki grafik, dizi erişimlerini koyu gri renkte gösterir (yine solda sanal ve sağda fiziksel ile); En üstteki grafik, sayfa tablosu belleğine erişimleri açık gri renkte gösterir (bu örnekteki sayfa tablosu fiziksel bağlamda bulunduğundan, yalnızca fizikseldir)). X ekseni, tüm izleme için, döngünün ilk beş yinelemesindeki bellek erişimlerini gösterir; döngü başına 10 bellek erişimi vardır; dört talimat getirme, bir açık bellek güncelleştirmesi ve bu dört getirmeyi çevirmek için beş sayfa tablosu erişimi ve bir açık güncelleştirme içerir.

Bu visualizasyonda ortaya çıkan kalıpları anlamlandırıp anlamlandıramayacağınıza bakın. Özellikle, döngü bu ilk beş yinelemenin ötesine geçmeye devam ettikçe neler değişecek? Hangi yeni bellek konumlarına erişilecek? Bunu anlayabiliyor musunuz ?

Bu, enbasit örnekler olmuştur (sadece birkaç satır C kodu) ve yine de gerçek uygulamaların gerçek bellek davranışını anlamanın karmaşıklığını zaten hissedebilirsiniz. Endişelenmeyin: meydan okurcasına daha da kötüye gidiyor, çünkü tanıtmak üzere olduğumuz mekanizmalar zaten karmaşık olan bu makineyi yalnızca karmaşıklaştırıyor .

18.6 Özet

Belleği sanallaştırma konusundaki zorluğumuza bir çözüm olarak sayfalama**(paging)** kavramını tanıttık. Disk belleğinin önceki yaklaşımlara (segmentasyon gibi) göre birçok avantajı vardır. İlk olarak, disk belleği (tasarım gereği) belleği sabit boyutlu birimlere böldüğü için harici parçalanmaya yol açmaz. İkincisi, oldukça esnektir ve sanal reklam elbisesi spaclarının seyrek kullanımını sağlar .

Bununla birlikte, disk belleği desteğinin özen gösterilmeden uygulanması, daha yavaş bir makineye (sayfa tablosuna erişmek için birçok ekstra bellek erişimiyle) ve bellek israfına (yararlı uygulama verileri yerine sayfa tablolarıyla dolu bellekle) yol açacaktır . Bu nedenle, sadece işe yaramakla kalmayıp aynı zamanda iyi çalışan bir çağrı sistemi bulmak için biraz daha fazla düşünmemiz gerekecek. Neyse ki, önümüzdeki iki bölüm bize bunu nasıl yapacağımızı gösterecek .

[3] Gerçekten üzgün değiliz. Ancak, üzgün olmadığımız için üzgünüz, eğer bu mantıklıysa.

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. Pay attention to "Volume 3A: System Programming Guide Part 1" and "Volume 3B: System Programming Guide Part 2".

[L78] he 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

Ödev (Simülasyon)

Bu ödevde, sanaldan fiziksele adres çevirisinin doğrusal sayfa tablolarıyla ne kadar basit çalıştığını anlayıp anlamadığınızı görmek için paging-linear-translate.py olarak bilinen basit bir program kullanacaksınız. Ayrıntılar için README'ye bakın.

Soru

1. Herhangi bir çeviri yapmadan önce, doğrusal sayfa tablolarının farklı parametreler verilen boyutu nasıl değiştirdiğini incelemek için simülatörü kullanalım. Farklı parametreler değiştikçe doğrusal sayfa tablolarının boyutunu hesaplayın. Önerilen bazı girişler aşağıdadır; -v seçeneğini kullanarak, kaç sayfa tablosu girişinin doldurulduğunu görebilirsiniz. İlk olarak, adres alanı büyüdükçe doğrusal sayfa tablosu boyutunun nasıl değiştiğini anlamak için şu 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 eğilimleri düşünmeye çalışın. Adres alanı büyüdükçe sayfa tablosu boyutu nasıl değişmelidir? Sayfa boyutu büyüdükçe? Neden genel olarak büyük sayfalar kullanmıyorsunuz?

2. Şimdi bazı çeviriler yapalım. Bazı küçük örneklerle başlayın ve adres alanına ayrılan sayfa sayısını -u bayrağıyla 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 adres alanında tahsis edilen sayfaların yüzdesini artırdığınızda ne olur?

3. Şimdi çeşitlilik için bazı farklı rastgele tohumlar ve bazı farklı (ve bazen oldukça çılgınca) adres alanı parametreleri 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 hangisi gerçekçi değildir? Neden?

4. Diğer bazı sorunları denemek için programı kullanın. Programın artık çalışmadığı yerlerin sınırlarını bulabilir misiniz? Örneğin, adres alanı boyutu fiziksel memory'den *büyükse* ne olur?

CEVAPLAR

1.SORU

-P 1k -a 1m -p 512m -v -n 0

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
ARG seed 0
ARG address space size 1m
ARG phys mem size 512m
ARG page size 1k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x8006104a
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x80033d4e
```

```
  [    1019]  0x8002e9c9
  [    1020]  0x00000000
  [    1021]  0x00000000
  [    1022]  0x00000000
  [    1023]  0x00000000

Virtual Address Trace

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

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

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 1k -a 2m -p 512m -v -n 0
ARG seed 0
ARG address space size 2m
ARG phys mem size 512m
ARG page size 1k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x8006104a
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x80033d4e
```

```
  [    2046]  0x8000eedd
  [    2047]  0x00000000

Virtual Address Trace

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

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

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 1k -a 4m -p 512m -v -n 0
ARG seed 0
ARG address space size 4m
ARG phys mem size 512m
ARG page size 1k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x8006104a
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x80033d4e
```

```
  [    4094]  0x00000000
  [    4095]  0x8002e298

Virtual Address Trace

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

1.Soru Açıklama: Yukardaki örneklerde de görüldüğü gibi 1m 2m ve 4m şeklinde artan örnekler var.

'-P' ve '-a' seçenekleri, komutun çalıştırılması sırasında kullanılacak bellek miktarlarını belirler.

'-p' (küçük p) bellek sayfalamasını gerçekleştirir.

'-v' seçeneği komutun çalışması sırasında daha fazla bilgi gösterilmesini sağlar.

Son olarak '-n' seçeneği komutun kaç kez tekrar edileceğini belirtir. Bu 3 örnekte de n seçeneği 0 olduğundan dolayı komut 1 defa çalıştı.

Spesifik olarak -a seçeneği adres uzayı boyutunu belirler. Bu komutlarda -a seçeneği ile çalışacağız. Bu 3 komutta sadece -a yani kullanılacak bellek miktarları değişiyor. İlk seçenekte 1m bu komutun çıktısında 1024 satır girdi gösterildi. 2m yaptığımızda 2048 satır girdi gösterildi. 4m yaptığımızda ise 4096 satır girdi gösterildi.

1.Sorunun ikinci kısmında ise sadece -P seçeneğini değiştireceğiz. Bu komutların çıktıları aşağıdaki sayfada bulunuyor.
-P seçeneği büyüdükçe 1k, 2k ve 4k şeklinde çıktının satır sayısı azalıyor. Aşağıdaki örneklerde de görüldüğü üzere:
1k 1024 satır sonuç veriyor
2k 512 satır sonuç veriyor
4k 256 satır sonuç veriyor

-P 1k -a 1m -p 512m -v -n 0

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 1k -a 1m -p 512m -v -n 0
ARG seed 0
ARG address space size 1m
ARG phys mem size 512m
ARG page size 1k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x8006104a
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x8003344e
  [    1019]  0x8002e9c9
  [    1020]  0x00000000
  [    1021]  0x00000000
  [    1022]  0x00000000
  [    1023]  0x00000000

Virtual Address Trace

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

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

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 2k -a 1m -p 512m -v -n 0
ARG seed 0
ARG address space size 1m
ARG phys mem size 512m
ARG page size 2k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x80030825
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x80019ea7
  [     510]  0x00000000
  [     511]  0x00000000

Virtual Address Trace

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

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

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 4k -a 1m -p 512m -v -n 0
ARG seed 0
ARG address space size 1m
ARG phys mem size 512m
ARG page size 4k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x80018412
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x8000cf53
  [     254]  0x80019c37
  [     255]  0x8001fb27

Virtual Address Trace

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

2.SORU

-P 1k -a 16k -p 32k -v -u 0

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 0
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x00000000
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x00000000
  [       4]  0x00000000
  [       5]  0x00000000
  [       6]  0x00000000
  [       7]  0x00000000
  [       8]  0x00000000
  [       9]  0x00000000
  [      10]  0x00000000
  [      11]  0x00000000
  [      12]  0x00000000
  [      13]  0x00000000
  [      14]  0x00000000
  [      15]  0x00000000

Virtual Address Trace
  VA 0x00003a39 (decimal:    14905) --> PA or invalid address?
  VA 0x00003ee5 (decimal:    16101) --> PA or invalid address?
  VA 0x000033da (decimal:    13274) --> PA or invalid address?
  VA 0x000039bd (decimal:    14781) --> PA or invalid address?
  VA 0x000013d9 (decimal:     5081) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

-P 1k -a 16k -p 32k -v -u 25

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 25
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x80000018
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x00000000
  [       4]  0x00000000
  [       5]  0x80000009
  [       6]  0x00000000
  [       7]  0x00000000
  [       8]  0x80000010
  [       9]  0x00000000
  [      10]  0x80000013
  [      11]  0x00000000
  [      12]  0x8000001f
  [      13]  0x8000001c
  [      14]  0x00000000
  [      15]  0x00000000

Virtual Address Trace
  VA 0x00003986 (decimal:    14726) --> PA or invalid address?
  VA 0x00002bc6 (decimal:    11206) --> PA or invalid address?
  VA 0x00001e37 (decimal:     7735) --> PA or invalid address?
  VA 0x00000671 (decimal:     1649) --> PA or invalid address?
  VA 0x00001bc9 (decimal:     7113) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

-P 1k -a 16k -p 32k -v -u 50

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 50
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x80000018
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x8000000c
  [       4]  0x80000009
  [       5]  0x00000000
  [       6]  0x8000001d
  [       7]  0x80000013
  [       8]  0x00000000
  [       9]  0x8000001f
  [      10]  0x8000001c
  [      11]  0x00000000
  [      12]  0x8000000f
  [      13]  0x00000000
  [      14]  0x00000000
  [      15]  0x80000008

Virtual Address Trace
  VA 0x00003385 (decimal:    13189) --> PA or invalid address?
  VA 0x0000231d (decimal:     8989) --> PA or invalid address?
  VA 0x000000e6 (decimal:      230) --> PA or invalid address?
  VA 0x00002e0f (decimal:    11791) --> PA or invalid address?
  VA 0x00001986 (decimal:     6534) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

-P 1k -a 16k -p 32k -v -u 75

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 75
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x80000018
  [       1]  0x80000008
  [       2]  0x8000000c
  [       3]  0x80000009
  [       4]  0x80000012
  [       5]  0x80000010
  [       6]  0x8000001f
  [       7]  0x8000001c
  [       8]  0x80000017
  [       9]  0x80000015
  [      10]  0x80000003
  [      11]  0x80000013
  [      12]  0x8000001e
  [      13]  0x8000001b
  [      14]  0x80000019
  [      15]  0x80000000

Virtual Address Trace
  VA 0x00002e0f (decimal:    11791) --> PA or invalid address?
  VA 0x00001986 (decimal:     6534) --> PA or invalid address?
  VA 0x000034ca (decimal:    13514) --> PA or invalid address?
  VA 0x00002ac3 (decimal:    10947) --> PA or invalid address?
  VA 0x00000012 (decimal:       18) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

-P 1k -a 16k -p 32k -v -u 100

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 1k -a 16k -p 32k -v -u 100
ARG seed 0
ARG address space size 16k
ARG phys mem size 32k
ARG page size 1k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x80000018
  [       1]  0x80000008
  [       2]  0x8000000c
  [       3]  0x80000009
  [       4]  0x80000012
  [       5]  0x80000010
  [       6]  0x8000001f
  [       7]  0x8000001c
  [       8]  0x80000017
  [       9]  0x80000015
  [      10]  0x80000003
  [      11]  0x80000013
  [      12]  0x8000001e
  [      13]  0x8000001b
  [      14]  0x80000019
  [      15]  0x80000000

Virtual Address Trace
  VA 0x00002e0f (decimal:    11791) --> PA or invalid address?
  VA 0x00001986 (decimal:     6534) --> PA or invalid address?
  VA 0x000034ca (decimal:    13514) --> PA or invalid address?
  VA 0x00002ac3 (decimal:    10947) --> PA or invalid address?
  VA 0x00000012 (decimal:       18) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

2.SORU AÇIKLAMA: Yukarda da görüldüğü gibi -P 1k -a 16k -p 32k -v -u ... sadece ... kısmını değiştirerek sonuçları inceledik.

Yukardaki ekran görüntülerinde de görüleceği üzere her seferinde 16 satır girdi gösteriliyor. Çünkü -P ya da -a seçeneğiyle ilgili herhangi bir değişiklik yapılmıyor.

Burada değiştirdiğimiz -u seçeneği kullanılacak işlem boyutu yüzdesini verir. Yani 0-100 arasında değerler verilmedir.

İşlem boyutumuz burada -p 32k olarak verilmiş. Bu da şu anlama geliyor: Eğer -u seçeneğini 100 yaparsak 32k nın tamamını kullanır.

3.SORU

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

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 8 -a 32 -p 1024 -v -s 1
ARG seed 1
ARG address space size 32
ARG phys mem size 1024
ARG page size 8
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x00000000
  [       1]  0x80000061
  [       2]  0x00000000
  [       3]  0x00000000

Virtual Address Trace
  VA 0x0000000e (decimal:       14) --> PA or invalid address?
  VA 0x00000014 (decimal:       20) --> PA or invalid address?
  VA 0x00000019 (decimal:       25) --> PA or invalid address?
  VA 0x00000003 (decimal:        3) --> PA or invalid address?
  VA 0x00000000 (decimal:        0) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

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

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 8k -a 32k -p 1m -v -s 2
ARG seed 2
ARG address space size 32k
ARG phys mem size 1m
ARG page size 8k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x80000079
  [       1]  0x00000000
  [       2]  0x00000000
  [       3]  0x8000005e

Virtual Address Trace
  VA 0x000055b9 (decimal:    21945) --> PA or invalid address?
  VA 0x00002771 (decimal:    10097) --> PA or invalid address?
  VA 0x00004d8f (decimal:    19855) --> PA or invalid address?
  VA 0x00004dab (decimal:    19883) --> PA or invalid address?
  VA 0x00004a64 (decimal:    19044) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

-P 1m -a 256m -p 512m -v -s 3

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 1m -a 256m -p 512m -v -s 3
ARG seed 3
ARG address space size 256m
ARG phys mem size 512m
ARG page size 1m
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x00000000
  [       1]  0x800000bd
  [       2]  0x80000140
  [       3]  0x00000000
  [     254]  0x80000159
  [     255]  0x00000000

Virtual Address Trace
  VA 0x0308b24d (decimal: 50901581) --> PA or invalid address?
  VA 0x042351e6 (decimal: 69423590) --> PA or invalid address?
  VA 0x02feb67b (decimal: 50247291) --> PA or invalid address?
  VA 0x0b46977d (decimal: 189175677) --> PA or invalid address?
  VA 0x0dbcceb4 (decimal: 230477492) --> PA or invalid address?

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

3.SORU AÇIKLAMA: İlk iki seçenekte -P seçeneği küçük olduğundan dolayı 4 er satır çıktıyla sunuçlanmış. Üçüncü komutta ise -P 1m e çıkarıldığı için 256 satır çıktı verilmiş.

3. Soruda kullandığımız -s seçeneği rastgele bir seme üretir. Bu seçenek rastgele sayıların üretilmesinde kullanılır. Seme; yanında verilen 1,2,3… gibi sayılardan daha önce üretilmiş ratgele sayıları çıktı olarak verir. Yani aynı komutu tekrar kullandığımızda tekrardan aynı değerleri ekrana basacaktır. Yeni değerler üretmeyecektir.

4.SORU

NOT: Girilen k ve m değerleri 2'nin katları olmalı

-P 1k -a 1m -p 1024m -v -n 0

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 1k -a 1m -p 1024m -v -n 0
ARG seed 0
ARG address space size 1m
ARG phys mem size 1024m
ARG page size 1k
ARG verbose True
ARG addresses -1

Error: must use smaller sizes (less than 1 GB) for this simulation.
```

-P 1k -a 512m -p 512m -v -n 0

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 1k -a 512m -p 512m -v -n 0
ARG seed 0
ARG address space size 512m
ARG phys mem size 512m
ARG page size 1k
ARG verbose True
ARG addresses -1

Error: physical memory size must be GREATER than address space size (for this simulation)
```

-P 1024k -a 1m -p 512m -v -n 0

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 1024k -a 1m -p 512m -v -n 0
ARG seed 0
ARG address space size 1m
ARG phys mem size 512m
ARG page size 1024k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)
  [       0]  0x80000184

Virtual Address Trace

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).

kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 1024k -a 1m -p 512m -v -n 0
```

-P 2048k -a 1m -p 512m -v -n 0

```
kd@ubuntu:~/ostep-homework/vm-paging$ python paging-linear-translate.py -P 2048k -a 1m -p 512m -v -n 0
ARG seed 0
ARG address space size 1m
ARG phys mem size 512m
ARG page size 2048k
ARG verbose True
ARG addresses -1


The format of the page table is simple:
The high-order (left-most) bit is the VALID bit.
  If the bit is 1, the rest of the entry is the PFN.
  If the bit is 0, the page is not valid.
Use verbose mode (-v) if you want to print the VPN # by
each entry of the page table.

Page Table (from entry 0 down to the max size)

Virtual Address Trace

For each virtual address, write down the physical address it translates to
OR write down that it is an out-of-bounds address (e.g., segfault).
```

4.SORU AÇIKLAMA: 4. Soruda bizden istenen programın limitlerini bulmamız.
Öncelikle programda -P,-a,-p gibi seçeneklere vereceğimiz değerler 2 nin katları olmalı.

2. olarak -P,-a,-p seçeneklerini doğru seçmemiz gerekli çünkü simülasyona ayırdığımız maximum bellek miktarını geçebilir. Program hata verir yukarda görüldüğü gibi.

Örneğin -P 2048k -a 1m -p 512m -v -n 0 komutu yerine
-P 1024k şeklinde çalıştırırsak komutu, bir çıktı elde ederiz ve bu çıktı 262.144 satır olacaktır. Ama 2048k olarak çalıştırınca program hata verecektir.