18

Sayfalama: Giriş

Bazen, herhangi bir alan yönetimi sorununu çözerken işletim sisteminin iki yaklaşımdan birini aldığı söylenir. Ne yazık ki, bu çözümün kendine has zorlukları var. Özellikle, bir alanı farklı büyüklükteki parçalara bölerken, alanın kendisi parçalanabilir **(fragmented)**, ve bu nedenle zaman içinde tahsis daha zor hale gelir.

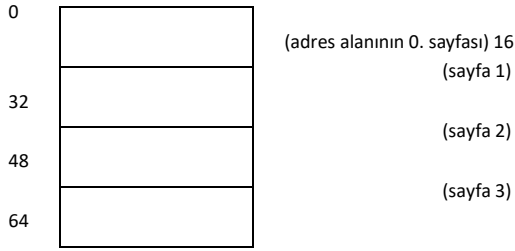
Bu nedenle, ikinci yaklaşımı dikkate almaya değer olabilir: alanı sabit boyutlu parçalara bölmek. Sanal bellekte buna fikir sayfalama **(paging)**, diyoruz ve eski ve önemli bir sisteme, Atlas'a [KE+62, L78] geri dönüyor. Bir işlemin adres alanını belirli sayıda değişken boyutlu mantıksal segmentlere (örneğin kod, yığın, yığın) bölmek yerine, onu her biri bir sayfa**(page)** olarak adlandırdığımız sabit boyutlu birimlere böleriz. Buna uygun olarak, fiziksel belleği, sayfa çerçeveleri adı verilen sabit boyutlu yuvalar dizisi olarak görüyoruz; bu çerçevelerin **(page frames**) her biri tek bir sanal bellek sayfası içerebilir. Bizim meydan okumamız:

CRUX: HAFIZA SAYFALARLA NASIL SANALLAŞTIRILIR
Segmentasyon sorunlarından kaçınmak için belleği sayfalarla nasıl sanallaştırabiliriz? Temel teknikler nelerdir? Minimum alan ve zaman giderleri ile bu tekniklerin iyi çalışmasını nasıl sağlarız?

18.1 Basit Bir Örnek ve Genel Bakış

Bu yaklaşımı daha açık hale getirmeye yardımcı olmak için basit bir örnekle açıklayalım. Şekil 18.1 (sayfa 2), dört adet 16 baytlık sayfa (sanal sayfa 0, 1, 2 ve 3) ile toplam olarak yalnızca 64 bayt boyutunda küçük bir adres alanı örneğini sunar. Gerçek adres alanları elbette çok daha büyüktür, genellikle 32 bit ve dolayısıyla 4 GB adres alanı, hatta 64 bittir[1]; kitapta, hazmetmeyi kolaylaştırmak için genellikle küçük örnekler kullanacağız.

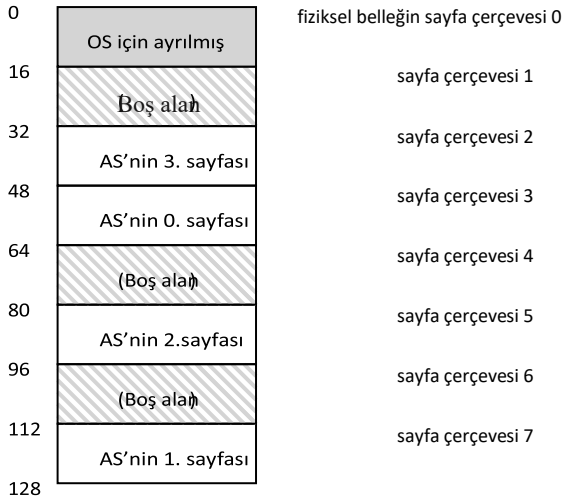
[1]64 bitlik bir adres alanını hayal etmek zor, inanılmaz derecede büyük. Bir benzetme yardımcı olabilir: 32 bitlik bir adres alanını bir tenis kortu büyüklüğünde düşünürseniz, 64 bitlik bir adres alanı yaklaşık Avrupa(!) boyutundadır.



Şekil 18.1: **Basit 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, bu da gülünç derecede küçüktür). Şemada da görebileceğiniz gibi, sanal adres alanının sayfaları fiziksel bellek boyunca farklı konumlara yerleştirilmiştir; diyagram ayrıca, kendisi için bir miktar fiziksel bellek kullanan işletim sistemini de gösterir.

Sayfalama, göreceğimiz gibi, önceki yaklaşımlarımıza göre bir takım avantajlara sahiptir. Muhtemelen en önemli gelişme esneklik olacaktır: tam gelişmiş bir sayfalama yaklaşımıyla sistem, bir sürecin adres alanını nasıl kullandığından bağımsız olarak bir adres alanının soyutlanmasını etkili bir şekilde destekleyebilecektir; örneğin, yığın ve yığının büyüdüğü yön ve bunların nasıl kullanıldığı hakkında varsayımlarda bulunmayacağız.



Şekil 18.2: **128 Baytlık Fiziksel Bellekte 64 Baytlık Adres Alanı**

Diğer bir avantaj, sayfalamanın sağladığı boş alan yönetiminin basitliğidir. Örneğin, işletim sistemi küçük 64 baytlık adres alanımızı sekiz

sayfalık fiziksel belleğimize yerleştirmek istediğinde, yalnızca dört boş sayfa bulur; belki de işletim sistemi bunun için tüm ücretsiz sayfaların ücretsiz bir listesini**(free list)** tutar ve bu listeden ilk dört ücretsiz sayfayı alır. Örnekte, işletim sistemi adres alanının (AS) sanal sayfası 0'ı fiziksel çerçeve 3'e, AS'nin sanal sayfası 1'i 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 veri yapısını tutar. Sayfa tablosunun ana rolü, adres alanının sanal sayfalarının her biri için adres çevirilerini **(address translations)** depolamak ve böylece her sayfanın fiziksel bellekte nerede olduğunu bize bildirmektir. Basit örneğimiz için (Şekil 18.2, sayfa 2), sayfa tablosu şu dört girdiye sahip olacaktır: (Sanal Sayfa 0 → Fiziksel Çerçeve 3), (VP 1 → PF 7), (VP 2 → PF 5) ve (VP 3 → PF 2).

Bu sayfa tablosunun süreç başına bir veri yapısı olduğunu hatırlamak önemlidir (tartıştığımız çoğu sayfa tablosu yapısı süreç başına yapılardır; değineceğimiz bir istisna, ters çevrilmiş sayfa tablosudur **(inverted page table**)). Yukarıdaki örneğimizde başka bir işlem yürütülecek olsaydı, sanal sayfaları açıkça farklı fiziksel sayfalarla eşleştiğinden (herhangi bir paylaşım modulo) işletim sisteminin bunun için farklı bir sayfa tablosu yönetmesi gerekirdi.

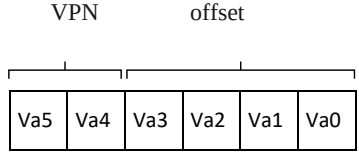
Artık bir adres-çeviri örneği yapacak kadar biliyoruz. Bu küçük adres alanına (64 bayt) sahip işlemin bir bellek erişimi gerçekleştirdiğini hayal edelim: movl <sanal adres>, %eax

Spesifik olarak, <sanal adres> adresinden kayıt eax'ına verilerin açık bir şekilde yüklenmesine dikkat edelim (ve böylece daha önce gerçekleşmiş olması gereken talimat getirmeyi görmezden gelelim).

İşlemin oluşturduğu bu sanal adresi çevirmek **(translate)** için önce onu iki bileşene ayırmamız gerekir: sanal sayfa numarası (VPN) **(virtual page number (VPN))** ve sayfa içindeki uzaklık **(offset)**. Bu örnek için işlemin sanal adres alanı 64 bayt olduğu için sanal adresimiz için toplam 6 bit ihtiyacımız var (26 = 64). Böylece sanal adresimiz şu şekilde kavramsallaştırılabilir:

V a 5	V a 4	V a 3	V a 2	V a 1	V a 0

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

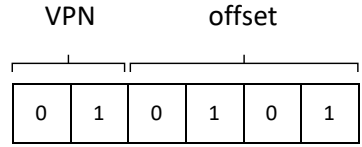


64 baytlık bir adres alanında sayfa boyutu 16 bayttır; bu yüzden 4 sayfa seçebilmemiz gerekiyor ve adresin en üstteki 2 biti tam da bunu yapıyor. Böylece 2 bitlik bir sanal sayfa numaramız (VPN) var. Kalan bitler bize sayfanın hangi baytıyla ilgilendiğimizi söyler, bu durumda 4 bit; buna ofset diyoruz.

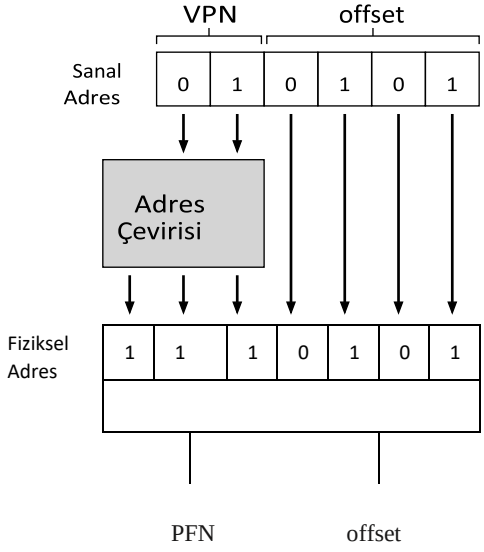
Bir işlem sanal bir adres oluşturduğunda, işletim sistemi ve donanım bunu anlamlı bir fiziksel adrese dönüştürmek için bir araya gelmelidir. Örneğin, yukarıdaki yükün sanal adres 21'e olduğunu varsayalım:

```
movl 21, %eax
```

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



Böylece, "21" sanal adresi "01" (veya 1) sanal sayfasının 5. ("0101"inci) baytındadır. Sanal sayfa numaramızla artık sayfa tablomuzu indeksleyebilir ve sanal sayfa 1'in hangi fiziksel çerçeve içinde olduğunu bulabiliriz. Yukarıdaki sayfa tablosunda fiziksel çerçeve numarası (PFN) **(physical frame number (PFN))** (bazen fiziksel sayfa numarası veya PPN (**physical page number** or **PPN**)) olarak da adlandırılır) 7'dir (ikili 111). Böylece, VPN'yi PFN ile değiştirerek bu sanal adresi çevirebilir ve ardından yükü fiziksel belleğe verebiliriz (Şekil 18.3).



Şekil 18.3**: Adres Çeviri Süreci**

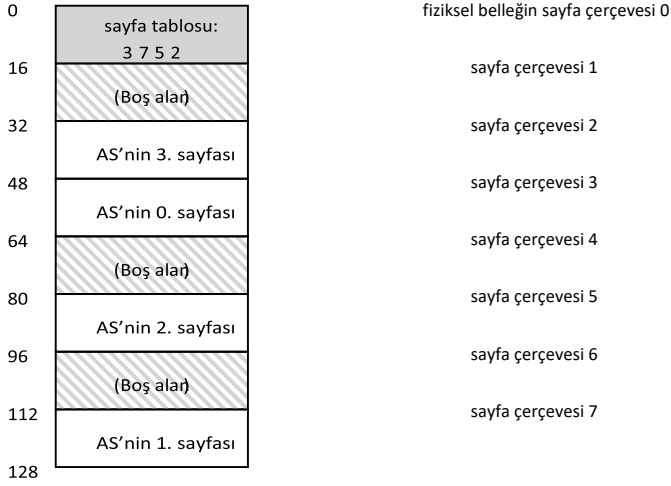


Figure Şekil 18.4: **Örnek: Çekirdek Fiziksel Belleğindeki Sayfa Tablosu**

Ofset aynı kalır (yani çevrilmez), çünkü ofset bize sayfada hangi baytı istediğimizi söyler. Nihai fiziksel adresimiz 1110101'dir (ondalık olarak 117) ve tam olarak yükümüzün veri getirmesini istediğimiz yerdir (Şekil 18.2, sayfa 2).

Bu temel genel bakışı göz önünde bulundurarak, artık sayfalama hakkında sahip olabileceğiniz birkaç temel soruyu sorabiliriz (ve umarız cevaplayabiliriz). Örneğin, bu sayfa tabloları nerede saklanıyor? Sayfa tablosunun tipik içerikleri nelerdir ve tablolar ne kadar büyük? Sayfalama sistemi (çok) yavaşlatıyor mu? Bu ve diğer aldatıcı sorular, en azından kısmen, aşağıdaki metinde yanıtlanmıştır. Okumaya devam et!

18.2 Sayfa Tabloları Nerede Saklanır?

Sayfa tabloları, daha önce tartıştığımız küçük bölüm tablosundan veya taban/sınır çiftinden çok daha büyük olabilir. Örneğin, 4 KB sayfaları olan tipik bir 32 bit adres alanı hayal edin. Bu sanal adres, 20 bitlik bir VPN ve 12 bitlik ofset olarak ikiye ayrılır (1 KB sayfa boyutu için 10 bitin gerekli olduğunu hatırlayın ve 4 KB'ye ulaşmak için sadece iki bit daha ekleyin).

20 bitlik bir VPN, işletim sisteminin her işlem için yönetmesi gereken 220 çeviri olduğu anlamına gelir (bu yaklaşık bir milyondur); fiziksel çeviriyi ve diğer yararlı şeyleri tutmak için sayfa tablosu girişi (PTE) **(page table entry (PTE))**başına 4 bayta ihtiyacımız olduğunu varsayarsak, her sayfa tablosu için gereken 4 MB'lık muazzam bir bellek elde ederiz!Bu oldukça büyük. Şimdi

çalışan 100 işlem olduğunu hayal edin: bu, işletim sisteminin tüm bu adres çevirileri için 400 MB belleğe ihtiyaç duyacağı anlamına gelir! Makinelerin gigabaytlarca belleğe sahip olduğu modern çağda bile,

KENARA: VERİ 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 sanaldan fiziksele adres çevirilerini saklar **(virtual-to-physical address translations)**, böylece sistemin bir adres alanındaki her sayfanın fiziksel bellekte gerçekte nerede olduğunu bilmesini sağlar.Her adres alanı bu tür çeviriler gerektirdiğinden, genellikle sistemde işlem başına bir sayfa tablosu bulunur. Sayfa tablosunun tam yapısı ya donanım tarafından belirlenir (eski sistemler) ya da işletim sistemi (modern sistemler) tarafından daha esnek bir şekilde yönetilebilir.

bunun büyük bir bölümünü yalnızca 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 ürkütücü olur ve belki de seni tamamen korkutur.

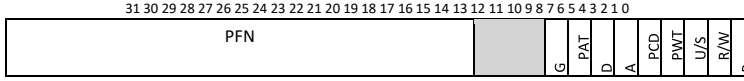
Sayfa tabloları çok büyük olduğu için, şu anda çalışan sürecin sayfa tablosunu depolamak için MMU'da herhangi bir özel çip üstü donanım tutmuyoruz. Bunun yerine, her işlem için sayfa tablosunu bellekte bir yerde saklarız. Şimdilik sayfa tablolarının işletim sisteminin yönettiği fiziksel bellekte yaşadığını varsayalım; daha sonra işletim sistemi belleğinin çoğunun sanallaştırılabileceğini ve dolayısıyla sayfa tablolarının işletim sistemi sanal belleğinde depolanabileceğini (ve hatta diske değiştirilebileceğini) göreceğiz, ancak bu şu anda çok kafa karıştırıcı, bu yüzden onu görmezden geleceğiz. Şekil 18.4'te (sayfa 5), işletim sistemi belleğindeki bir sayfa tablosunun resmi gösterilmektedir; 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 biçim, yalnızca bir dizi olan doğrusal sayfa tablosu **(linear page table)** olarak adlandırılır. İşletim sistemi, diziyi sanal sayfa numarasına (VPN) göre dizine ekler ve istenen fiziksel çerçeve numarasını (PFN) bulmak için bu dizinde sayfa tablosu girişine (PTE) bakar. Şimdilik bu basit lineer yapıyı kabul edeceğiz; sonraki bölümlerde, sayfalamayla ilgili bazı sorunları çözmeye yardımcı olması için daha gelişmiş veri yapılarından yararlanacağız.

Her bir PTE'nin içeriğine gelince, orada bir düzeyde anlaşılmaya değer birkaç farklı parçamız var. Belirli bir çevirinin geçerli olup olmadığını belirtmek için geçerli bir bit **(valid bit)** ortaktır; örneğin, bir program çalışmaya başladığında, adres alanının bir ucunda kod ve öbek, diğer ucunda yığın olacaktır. Aradaki

kullanılmayan tüm alan geçersiz olarak işaretlenecek ve işlem bu tür bir belleğe erişmeye çalışırsa, işletim sistemine muhtemelen işlemi sonlandıracak bir tuzak oluşturacaktır. Bu nedenle, geçerli bit, seyrek bir adres alanını desteklemek için çok önemlidir; adres alanındaki kullanılmayan tüm sayfaları geçersiz **(invalid)** olarak işaretleyerek, bu sayfalar için fiziksel çerçeve ayırma ihtiyacını ortadan kaldırır ve böylece büyük miktarda bellek tasarrufu sağlarız.



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

Ayrıca, sayfanın okunup okunamayacağını, yazılacağını veya yürütülebileceğini gösteren koruma bitlerimiz **(protection bits)** olabilir. Yine bu bitlerin izin vermediği bir sayfaya erişim, işletim sistemine tuzak oluşturacaktır.

Önemli olan birkaç başka bit daha var ama şimdilik fazla konuşmayacağız. Mevcut bir bit **(present bit)**, bu sayfanın fiziksel bellekte mi yoksa diskte mi olduğunu gösterir (yani değiştirilmiş **(swapped out)**). Fiziksel bellekten daha büyük olan adres alanlarını desteklemek için adres alanının parçalarının diske nasıl değiştirileceğini incelediğimizde bu mekanizmayı daha iyi anlayacağız; takas **(swap)**, işletim sisteminin nadiren kullanılan sayfaları diske taşıyarak fiziksel belleği boşaltmasına olanak tanır. Sayfanın belleğe alındığından beri değiştirilip değiştirilmediğini gösteren kirli bir bit **(dirty bit)** de yaygındır.

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

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

x86 çağrı desteği hakkında daha fazla ayrıntı için Intel Mimarisi Kılavuzlarını [I09] okuyun. Ancak önceden uyarılmalıdır; Bunlar gibi kılavuzları okumak, oldukça bilgilendirici olsa da (ve işletim sisteminde bu tür sayfa tablolarını kullanmak için kod yazanlar için kesinlikle gerekli), ilk başta zor olabilir. Biraz sabır ve çokça istek gerekiyor.

KENARA: NEDEN GEÇERLİ BİT YOK?

Intel örneğinde, ayrı geçerli ve mevcut bitlerin olmadığını, sadece mevcut bir bitin (P) olduğunu fark edebilirsiniz. Bu bit ayarlanmışsa (P=1), sayfanın hem mevcut hem de geçerli olduğu anlamına gelir. Değilse (P=0), bu, sayfanın bellekte bulunmadığı (ancak geçerli olduğu) veya geçerli olmadığı anlamına gelir. P=0

olan bir sayfaya erişim, işletim sistemine yönelik bir tuzağı tetikleyecektir; işletim sistemi daha sonra sayfanın geçerli olup olmadığını (ve bu nedenle belki de geri değiştirilmesi gerektiğini) veya olmadığını (ve dolayısıyla program belleğe yasa dışı bir şekilde erişmeye çalışıyor) belirlemek için sakladığı ek yapıları kullanmalıdır. Bu tür sağduyululuk, genellikle işletim sisteminin tam bir hizmet oluşturabileceği minimum özellik kümesini sağlayan donanımda yaygındır.

18.4 Sayfalama: Ayrıca Çok Yavaş

Hafızadaki sayfa tabloları ile bunların çok büyük olabileceğini zaten biliyoruz. Görünüşe göre, işleri de yavaşlatabilirler. Örneğin, basit talimatımızı alın:

```
movl 21, %eax
```

Yine, adres 21'e yapılan açık referansı inceleyelim ve talimat getirme konusunda endişelenmeyelim. Bu örnekte, çeviriyi bizim yerimize donanımın yaptığını varsayacağız. İstenen veriyi getirmek için sistem önce sanal adresi (21) doğru fiziksel adrese (117) çevirmelidir **(translate)**. Bu nedenle, 117 adresindeki verileri getirmeden önce, sistem önce işlemin sayfa tablosundan uygun sayfa tablosu girişini getirmeli, çeviriyi gerçekleştirmeli ve ardından verileri fiziksel bellekten yüklemelidir.

Bunu yapmak için, donanımın o anda çalışmakta olan işlem için sayfa tablosunun nerede olduğunu bilmesi gerekir. Şimdilik tek bir sayfa tablosu temel kaydının, sayfa tablosunun başlangıç konumunun fiziksel adresini içerdiğini varsayalım. İstenen PTE'nin yerini bulmak için donanım böylece aşağıdaki işlevleri yerine getirecektir:

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

Örneğimizde VPNMASK, tam sanal adresten VPN bitlerini seçen 0x30 (onaltılık 30 veya ikili 110000) olarak ayarlanır; SHIFT, doğru tamsayı sanal sayfa numarasını oluşturmak için VPN bitlerini aşağı taşıyacak şekilde 4'e (ofsetteki bit sayısı) ayarlanır. Örneğin, sanal adres 21 (010101) ile ve maskeleme bu değeri 010000'e dönüştürür; kaydırma onu 01'e veya isteğe göre sanal sayfa 1'e dönüştürür. Daha sonra bu değeri, sayfa tablosu temel kaydı tarafından işaret edilen PTE 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. Spesifik olarak, PFN'nin SHIFT tarafından sola kaydırıldığını ve ardından son adresi aşağıdaki gibi oluşturmak için ofset ile bit düzeyinde OR'lendiğini düşünebilirsiniz:

```
offset = VirtualAddress & OFFSET_MASK PhysAddr
= (PFN << SHIFT) | offset
```

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

Özetlemek gerekirse, şimdi her bellek referansında ne olduğuyla ilgili ilk protokolü açıklıyoruz. Şekil 18.6 (sayfa 9) yaklaşımı göstermektedir. Her bellek referansı için (bir talimat getirme veya açık bir yükleme veya depolama), sayfalama, önce sayfa tablosundan çeviriyi getirmek için fazladan bir bellek referansı gerçekleştirmemizi 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)
```

Figure 18.6: **Accessing Memory With Paging**

iş demek! Fazladan bellek referansları maliyetlidir ve bu durumda muhtemelen işlemi iki veya daha fazla kat yavaşlatacaktır.

Ve şimdi umarım çözmemiz gereken iki gerçek problem olduğunu görebilirsiniz. Hem donanımın hem de yazılımın dikkatli tasarımı olmadan, sayfa tabloları sistemin çok yavaş çalışmasına ve çok fazla bellek kaplamasına neden olur. 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 kullanılırken ortaya çıkan tüm bellek erişimlerini göstermek için şimdi basit bir bellek erişimi örneği üzerinden izleyeceğiz. İlgilendiğimiz kod parçacığı (C'de array.c adlı bir dosyada) aşağıdaki gibidir:

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

Array.c'yi derliyoruz ve aşağıdaki komutlarla çalıştırıyoruz:

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

Elbette, bu kod parçacığına (sadece bir diziyi başlatan) hangi belleğin erişeceğini gerçekten anlamak için, birkaç şey daha bilmemiz (veya varsaymamız) gerekecek. İlk olarak, diziyi bir döngüde başlatmak için hangi derleme yönergelerinin kullanıldığını görmek için (Linux'ta objdump veya Mac'te otool kullanarak) ortaya çıkan ikiliyi parçalarına **(disassemble)** ayırmamız gerekecek. İşte ortaya çıkan montaj kodu:

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

Biraz x86 biliyorsanız, kodu anlamak aslında oldukça kolaydır [Basitlik için her talimatın dört bayt boyutunda olduğunu varsayarak burada biraz hile yapıyoruz; gerçekte, x86 **(x86)** komutları değişken boyutludur.]. İlk talimat, sıfır değerini ($0x0 olarak gösterilir) dizinin konumunun sanal bellek adresine taşır; bu adres, %edi'nin içeriği alınarak ve buna %eax çarpı dört eklenerek hesaplanır. Bu nedenle, %edi dizinin temel adresini tutarken, %eax dizi indeksini (i) tutar; dizi, her biri dört bayt boyutunda bir tamsayılar dizisi olduğu için dört ile çarpıyoruz.

İkinci talimat, %eax'ta tutulan dizi indeksini artırır ve üçüncü talimat, bu kaydın içeriğini onaltılık değer 0x03e8 veya ondalık 1000 ile karşılaştırır. testler), dördüncü komut döngünün en üstüne atlar.

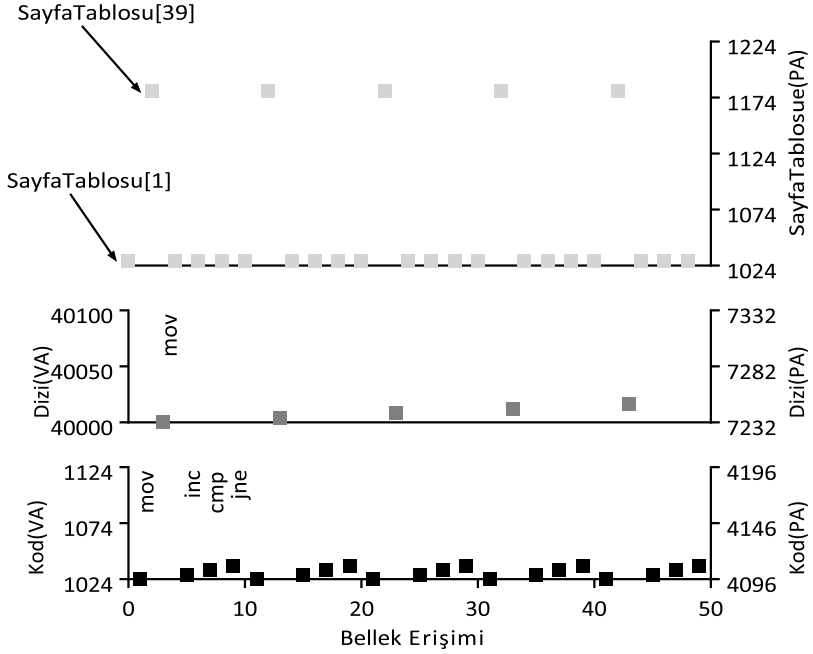
Bu komut dizisinin hangi belleğe eriştiğini anlamak için (hem sanal hem de fiziksel düzeylerde), sanal bellekte kod parçacığının ve dizinin nerede bulunduğu ve ayrıca sayfa tablosunun içeriği ve konumu hakkında bir şeyler varsaymamız gerekecek.

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

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

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

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



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

Artık programın bellek referanslarını izlemeye hazırız. Çalıştırıldığında, her talimat getirme işlemi iki bellek referansı oluşturur: biri talimatın içinde bulunduğu fiziksel çerçeveyi bulmak için sayfa tablosuna, diğeri ise onu işlenmek üzere CPU'ya getirmek için talimatın kendisine. Ek olarak, mov talimatı biçiminde bir açık bellek referansı vardır; bu, önce (dizi sanal adresini doğru fiziksel adrese çevirmek için) başka bir sayfa tablosu erişimi ve ardından dizinin kendisine erişim ekler.

İlk beş döngü yinelemesi için tüm süreç Şekil 18.7'de (sayfa 11) gösterilmektedir. En alttaki grafik, y eksenindeki komut belleği referanslarını siyah olarak gösterir (sanal adresler solda ve gerçek fiziksel adresler sağda); ortadaki grafik, dizi erişimlerini koyu gri renkte gösterir (yine solda sanal ve sağda fiziksel olarak); son olarak, en üstteki grafik, sayfa tablosu belleği erişimlerini açık gri renkte gösterir (bu örnekteki sayfa tablosu fiziksel bellekte bulunduğundan yalnızca fiziksel). Tüm izleme için x ekseni, döngünün ilk beş yinelemesindeki bellek erişimlerini gösterir; döngü başına 10 bellek erişimi vardır; bu, dört talimat getirme, bir açık bellek güncellemesi ve bu dört getirme ve bir açık güncellemeyi çevirmek için beş sayfa tablosu erişimi içerir.

Bu görselleştirmede ortaya çıkan kalıpları anlamlandırabilecek misiniz bir bakın. Özellikle, döngü bu ilk beş yinelemenin ötesinde çalışmaya devam ettikçe ne değişecek? Hangi yeni bellek konumlarına erişilecek? Anlayabilir misin?

Bu, örneklerin en basitiydi (yalnızca birkaç C kodu satırı) ve yine de gerçek uygulamaların gerçek bellek davranışını anlamanın karmaşıklığını şimdiden hissedebiliyor olabilirsiniz. Endişelenmeyin: kesinlikle daha da kötüye gidiyor, çünkü tanıtmak üzere olduğumuz mekanizmalar zaten karmaşık olan bu mekanizmayı daha da karmaşık hale getiriyor. Üzgünüm[Gerçekten üzgün değiliz. Ama, eğer bu mantıklıysa, üzgün olmadığımız için üzgünüz[2]!

18.6 Özet

Belleği sanallaştırma sorunumuz için bir çözüm olarak sayfalama **(paging)** kavramını tanıttık. Sayfalamanın önceki yaklaşımlara (segmentasyon gibi) göre birçok avantajı vardır. Birincisi, disk belleği (tasarım gereği) belleği sabit boyutlu birimlere böldüğü için harici parçalanmaya yol açmaz. İkincisi, oldukça esnektir ve sanal adres alanlarının seyrek olarak kullanılmasını sağlar.

Bununla birlikte, sayfalama desteğini dikkatsizce uygulamak, daha yavaş bir makineye (sayfa tablosuna erişmek için birçok ekstra bellek erişimiyle) ve ayrıca bellek israfına (kullanışlı uygulama verileri yerine sayfa tablolarıyla dolu bellekle) yol açacaktır. Bu nedenle, yalnızca çalışan değil, aynı zamanda iyi çalışan bir çağrı sistemi bulmak için biraz daha fazla düşünmemiz gerekecek. Neyse ki sonraki iki bölüm bize bunu nasıl yapacağımızı gösterecek.

Referanslar

[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.*

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

Ödev (Simülasyon)

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

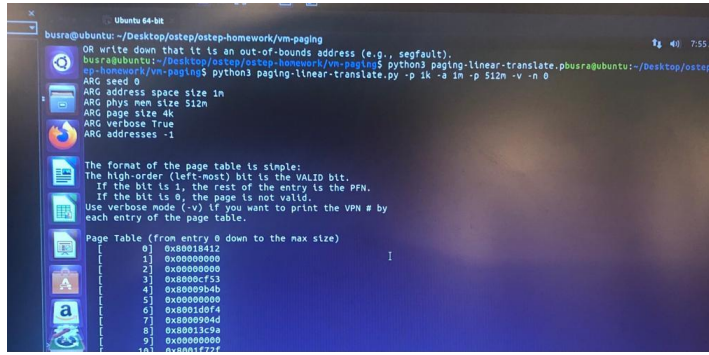
Sorular

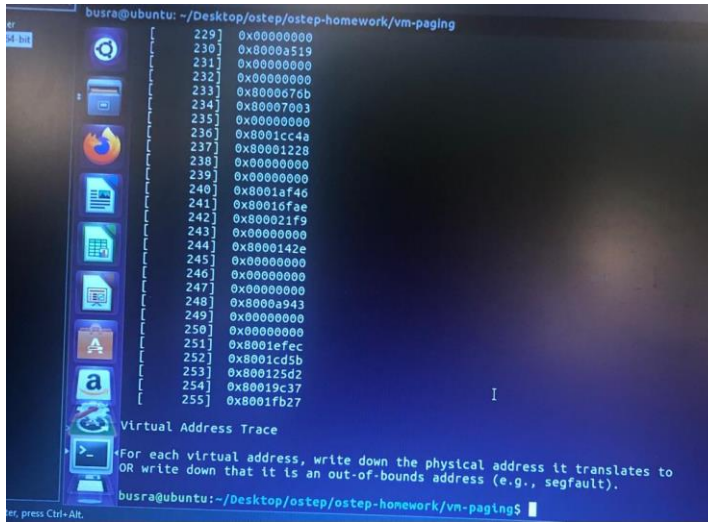
1. Herhangi bir çeviri yapmadan önce, farklı parametreler verildiğinde doğrusal sayfa tablolarının boyutunun nasıl değiştiğini incelemek için simülatörü kullanalım. Farklı parametreler değiştikçe doğrusal sayfa tablolarının boyutunu hesaplayın. Önerilen bazı girdiler aşağıdadır; -v işaretini kullanarak, kaç tane 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
```

Bizden sayfa tablosu büyüklüğünün nasıl değiştiğini anlamak için yukarıda 3 farklı şekilde deneyerek çalıştırmamız istendi. İlk olarak aşağıdaki girdiyle başlıyoruz ve elde ettiğimiz ekran görüntüsü de aşağıdaki gibidir:

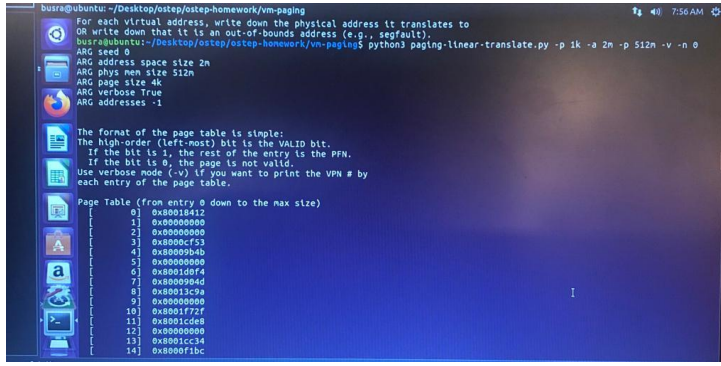
```
-P 1k -a 1m -p 512m -v -n 0
```

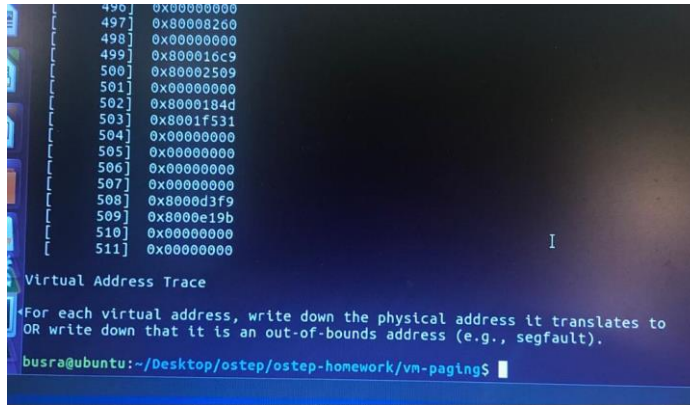




İkinci olarak aşağıdaki girdiyi çalıştırıyoruz elde ettiğimiz ekran görüntüsü de aşağıdaki gibidir:

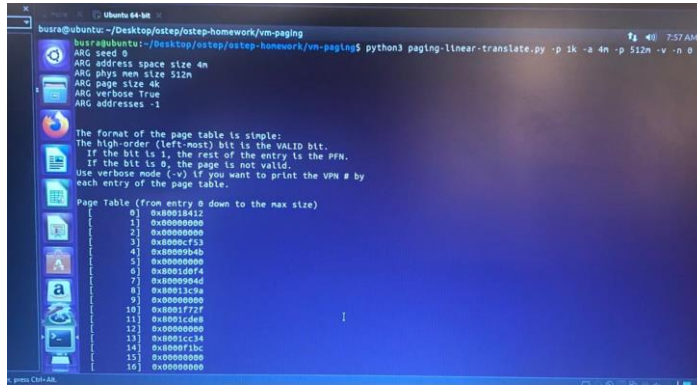
```
-P 1k -a 2m -p 512m -v -n 0
```

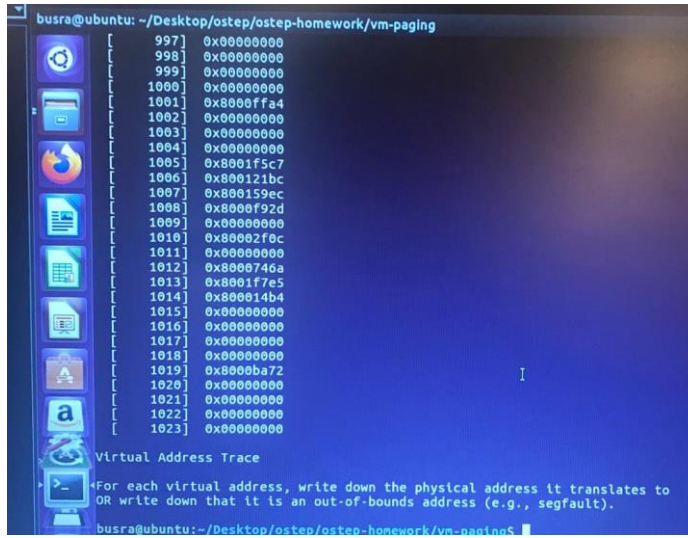




Üçüncü olarak aşağıdaki girdiyi çalıştırıyoruz ve elde ettiğimiz ekran görüntüsü de aşağıdaki gibidir:

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





Yukarıdaki çıktılara bakarak sadece m parametresinin değiştiğini görebiliriz. Bu doğrultuda m parametresi arttıkça çıktılarda da ciddi artışlar görülmektedir. Ayrıca -v flagi (bayrağı), hayatınızı kolaylaştırmak için VPN numaralarını yazdırır.

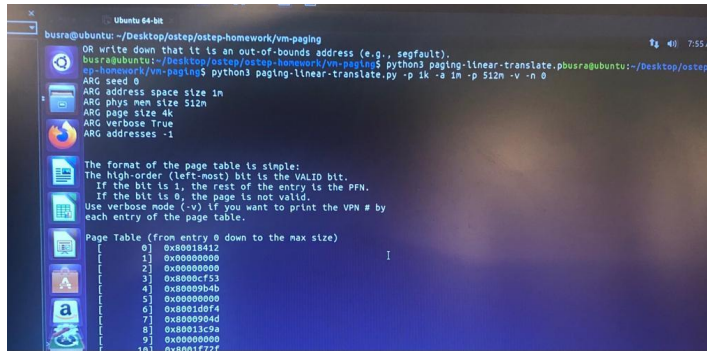
Ardından, sayfa boyutu büyüdükçe doğrusal sayfa tablosu boyutunun nasıl değiştiğini anlamak için:

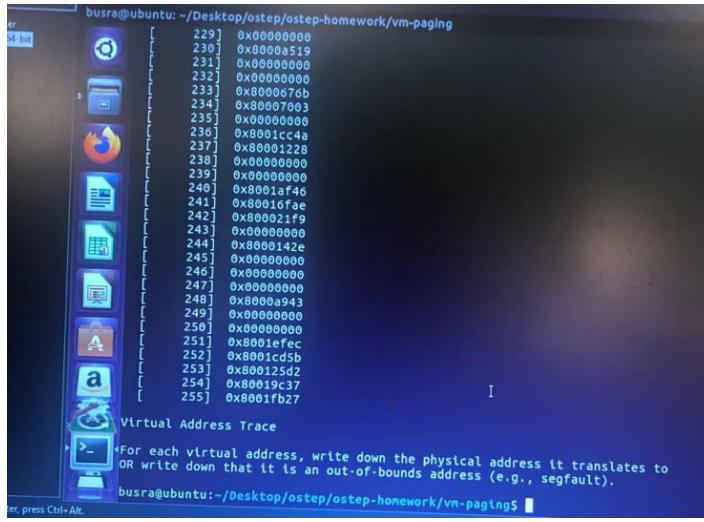
```
-P 1k -a 1m -p 512m -v -n 0
-P 2k -a 1m -p 512m -v -n 0
-P 4k -a 1m -p 512m -v -n 0
```

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

Bizden sayfa tablosu boyutunun nasıl değiştiğini anlamak için yukarıda 3 farklı şekilde deneyerek çalıştırmamız istendi. İlk olarak aşağıdaki girdiyle başlıyoruz ve elde ettiğimiz ekran görüntüsü de aşağıdaki gibidir:

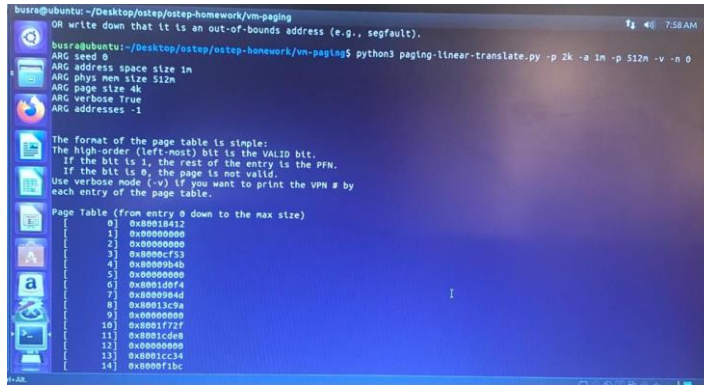
```
-P 1k -a 1m -p 512m -v -n 0
```

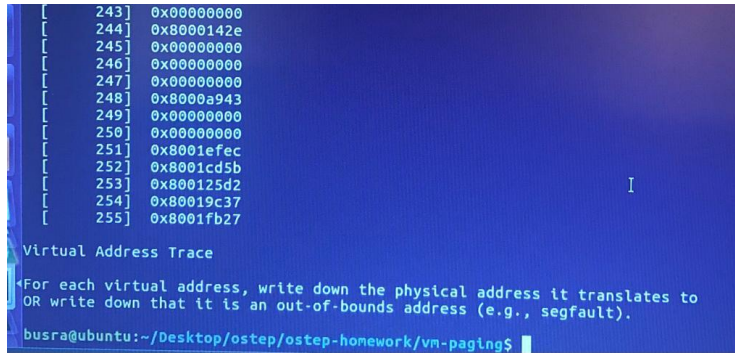




İkinci olarak aşağıdaki girdiyle başlıyoruz ve elde ettiğimiz ekran görüntüsü de aşağıdaki gibidir:

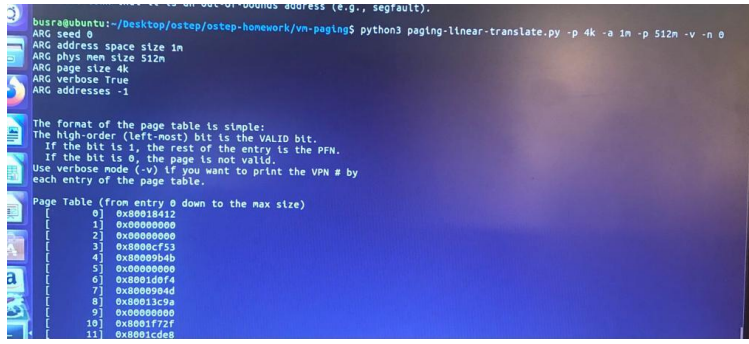
```
-P 2k -a 1m -p 512m -v -n 0
```

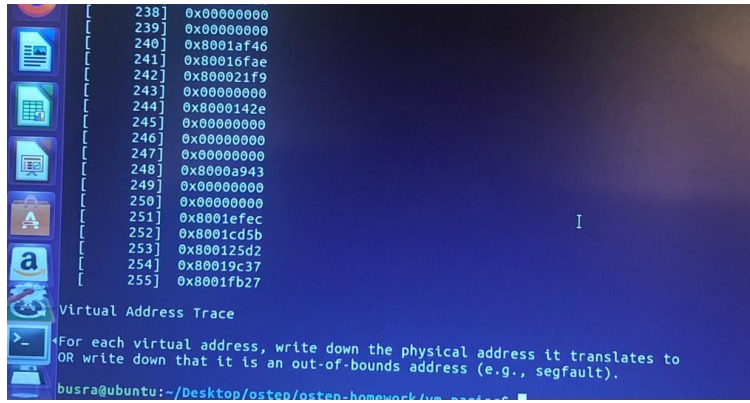




Üçüncü olarak aşağıdaki girdiyle başlıyoruz ve elde ettiğimiz ekran görüntüsü de aşağıdaki gibidir:

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





ARG page bize her seferinde 4k olarak bir çıktı veriyor.

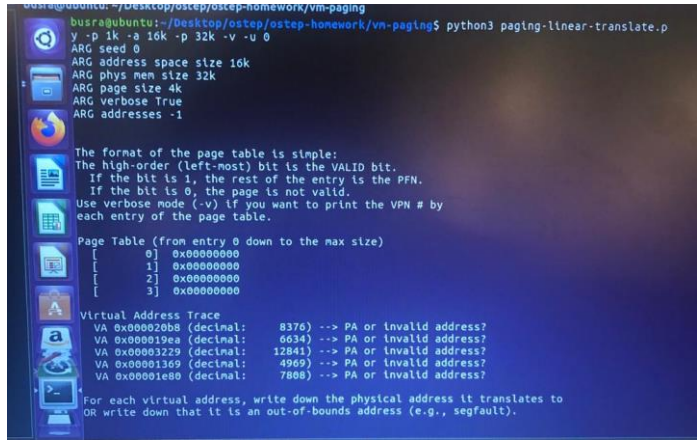
2. Şimdi bazı çeviriler yapalım. Bazı küçük örneklerle başlayın ve -u bayrağıyla adres alanına ayrılan sayfa sayısını değiştirin. Örneğin:

```
-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 alanına ayrılan sayfaların yüzdesini artırdığınızda ne olur?

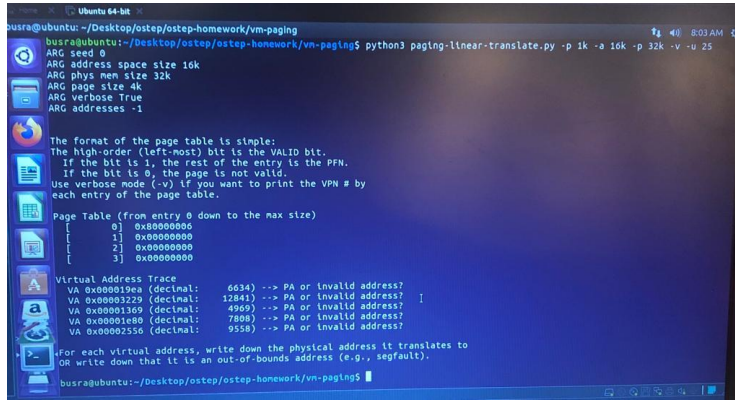
Bizden adres alanına ayrılan sayfa sayısını değiştirmemiz ve nasıl değiştiğini anlamak için yukarıda 5 farklı şekilde yüzdelerle deneyerek çalıştırmamız istendi. İlk olarak aşağıdaki girdiyle başlıyoruz, yüzde sıfırı çalıştırıyoruz ve elde ettiğimiz ekran görüntüsü de aşağıdaki gibidir:

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



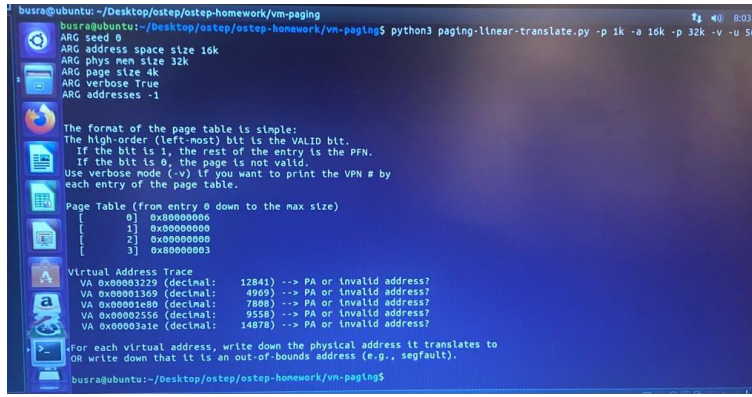
İkinci olarak aşağıdaki girdiyle başlıyoruz, yüzde yirmi beşi çalıştırıyoruz ve elde ettiğimiz ekran görüntüsü de aşağıdaki gibidir:

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



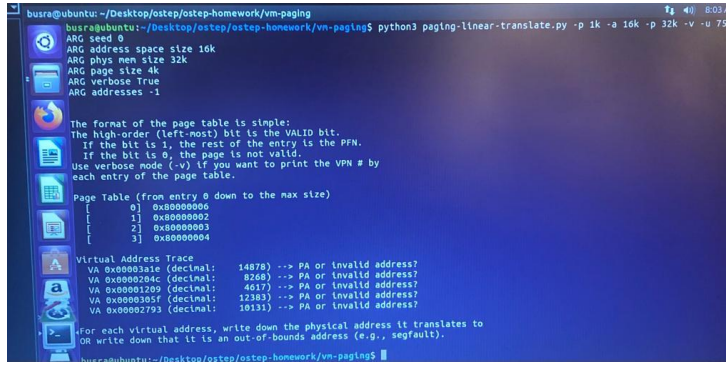
Üçüncü olarak aşağıdaki girdiyle başlıyoruz, yüzde elliyi çalıştırıyoruz ve elde ettiğimiz ekran görüntüsü de aşağıdaki gibidir:

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



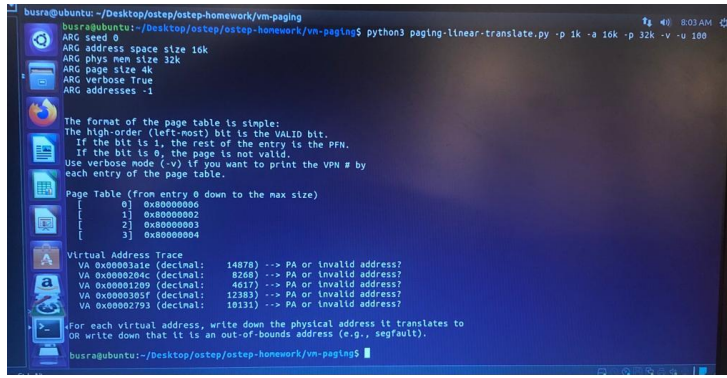
Dördüncü olarak aşağıdaki girdiyle başlıyoruz, yüzde yetmiş beşi çalıştırıyoruz ve elde ettiğimiz ekran görüntüsü de aşağıdaki gibidir:

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



Beşinci olarak aşağıdaki girdiyle başlıyoruz, yüzde yüzü çalıştırıyoruz ve elde ettiğimiz ekran görüntüsü de aşağıdaki gibidir:

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



Yaptığımız 5 farklı ekran görüntüsü sonucunda virtual adress trace içerisinde decimal değerlerinin değiştiği görülüyor. Ve doğru orantılı olarak yüzde arttıkça arttığını söyleyebiliriz. Ayrıca -u flagi (bayrağı), geçerli eşlemelerin oranını %0'dan (-u 0) %100'e (-u 100) kadar değiştirir. Varsayılan değer 50'dir; bu, sanal adres alanındaki sayfaların kabaca 1/2'sinin geçerli olacağı anlamına gelir.

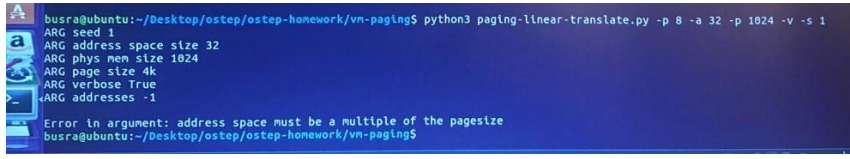
3. Şimdi çeşitlilik için bazı farklı rasgele tohumlar ve bazı farklı (ve bazen oldukça çılgınca) adres alanı parametrelerini deneyelim:

```
-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 hangisi gerçekçi değil? Neden? Niye?

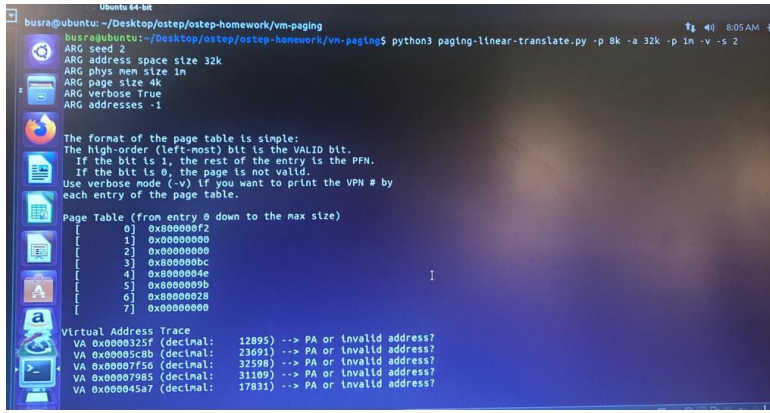
Bu soruda da daha farklı (birbirinden oldukça farklı) parametreleri deneyip onların çıktılarını almamız isteniyor. İlk olarak aşağıdaki parametreyle başlıyoruz çalıştırmaya ve elde ettiğimiz ekran görüntüsü de aşağıdaki gibidir:

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



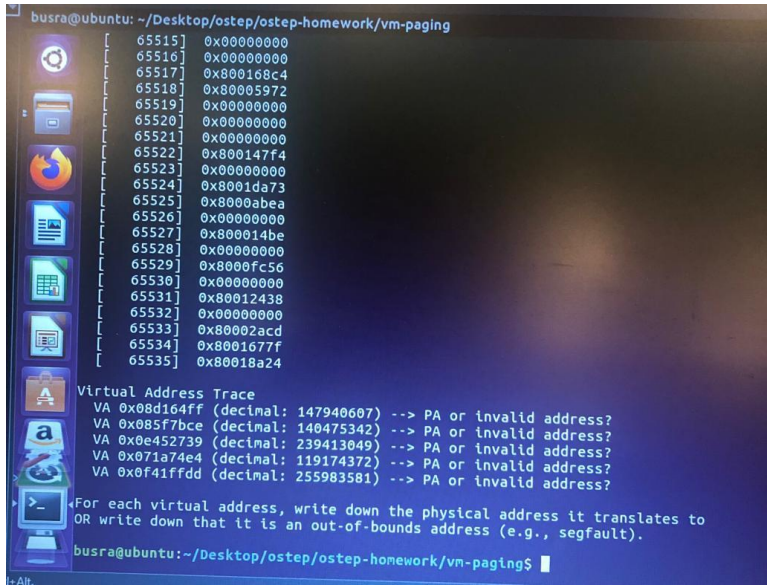
İkinci olarak aşağıdaki parametreyle başlıyoruz çalıştırmaya ve elde ettiğimiz ekran görüntüsü de aşağıdaki gibidir:

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



Üçüncü olarak aşağıdaki parametreyle başlıyoruz çalıştırmaya ve elde ettiğimiz ekran görüntüsü de aşağıdaki gibidir:

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



Yaptığımız 3 farklı ekran görüntüsü sonucunda virtual adress trace içerisinde decimal değerlerinin ve page table sayılarının değiştiği

görülüyor. Ve doğru orantılı olarak -s flagi arttıkça arttığını söyleyebiliriz. Ayrıca -s flagi (bayrağı), rastgele çekirdeği değiştirir ve böylece çevrilecek farklı sanal adreslerin yanı sıra farklı sayfa tablosu değerleri oluşturur.

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

Bellek bilgisayarı oluşturan 3 ana bileşenden biridir.İşlemcinin çalıştırdığı programlar ve programa ait bilgiler bellek üzerinde saklanır. Bellek geçici bir depolama alanıdır. Bellek üzerindeki bilgiler güç kesildiği anda kaybolurlar. Bu nedenle bilgisayarlarda programları daha uzun süreli ve kalıcı olarak saklamak için farklı birimler mevcuttur.
Fiziksel belleğe RAM adı verilir. Bu ad bellekte bir konuma rastgele ve hızlı bir şekilde erişebildiğimiz için verilmiştir. RAM'de sadece işlemcide çalışan program parçaları tutulur ve elektrik kesildiği anda RAM'deki bilgiler silinir.
Adres alanı ise bir program veya işlem için kullanılabilen bellekteki geçerli bir adres aralığıdır. Yani, bir programın veya işlemin erişebileceği bellektir. Bellek fiziksel veya sanal olabilir ve talimatları yürütmek ve veri depolamak için kullanılır.
Bir adres alanının boyutu, sanal bellek adı verilen bir bellek yönetimi tekniği kullanılarak fiziksel bellekten daha büyük yapılabilir. Fakat normal şartlarda büyük olması sıkıntı çıkarmasına sebep olur. Erişmek istediğimiz bir çok şeye biz farketmesek de engeller. İçten içe bilgisayarı tüketir. Bu nedenlerle dengeli olmasına dikkat etmeliyiz.

Örneğin `-P 1m -a 256m -p 512m -v -s 3` şu parametrede çok büyük çıktığı için sorguyu yazdığımız baş kısma kadar çıkamıyoruz.Yani bir süre sonra çalışmıyor.