20

Sayfalama: Daha Küçük Tablolar

Şimdi sayfalamanın getirdiği ikinci sorunu ele alıyoruz: sayfa tabloları çok büyük ve bu nedenle çok fazla bellek tüketiyor. Doğrusal bir sayfa tablosuyla başlayalım. Hatırlayabileceğiniz[1] gibi, doğrusal sayfa tabloları oldukça büyük. 4 KB (212 bayt) sayfa ve 4 bayt sayfa tablosu girişi ile yeniden 32 bitlik bir adres alanı (232 bayt) oluşur. Dolayısıyla, bir adres alanı içinde kabaca bir milyon sanal sayfa bulunur (232); sayfa-tablo giriş boyutu ile çarptığınızda, sayfa tablomuzun boyutunun 4 MB olduğunu görürsünüz. Ayrıca hatırlayın: genellikle sistemdeki her işlem için bir sayfa tablomuz olur! Yüz aktif işlemle (modern bir sistemde alışılmış değildir), yalnızca sayfa tabloları için yüzlerce megabayt bellek ayıracağız! Sonuç olarak, bu ağır yükü azaltmak için bazı teknikler arayışındayız. Onlardan çok var, hadi gidelim. Ama bizim dönüm noktamızdan önce değil:

MESELE: SAYFA TABLOLARI NASIL DAHA KÜÇÜK YAPILIR?
Basit dizi tabanlı sayfa tabloları (genellikle doğrusal sayfa tabloları olarak adlandırılır) çok büyük, tipik sistemlerde çok fazla bellek kaplıyor. Sayfa tablolarını nasıl küçültebiliriz? Anahtar fikirler neler? Bu yeni veri yapılarının bir sonucu olarak hangi verimsizlikler ortaya çıkıyor?

20.1 Basit Çözüm: Daha Büyük Sayfalar

Sayfa tablosunun boyutunu tek bir basit yolla azaltabiliriz: daha büyük sayfalar kullanın. 32 bit adres alanımızı tekrar alın, ancak bu sefer 16 KB sayfaları varsayalım. Böylece 18 bitlik bir VPN artı 14 bitlik bir kaymamız olur Her bir PTE için aynı boyutun (4 bayt) olduğu varsayılarak, artık lineer sayfa tablomuzda 2 girişimiz var ve bu nedenle sayfa tablosu başına toplam 1 MB boyut, bir faktör

[1]Ya da gerçekten, yapmayabilirsin; Bu çağrı olayı kontrolden çıkıyor, değil mi? Bununla birlikte, çözüme geçmeden önce çözdüğünüz sorunu anladığınızdan her zaman emin olun; aslında, eğer sorunu anlarsanız, genellikle çözümü kendiniz de elde edebilirsiniz. Burada sorun açık olmalı: basit doğrusal (dizi tabanlı) sayfa tabloları çok büyük.

KENARA: ÇOKLU SAYFA BOYUTLARI

Bir yandan, birçok mimarinin (ör. MIPS, SPARC, x86-64) artık birden çok sayfa boyutunu desteklediğini unutmayın. Genellikle küçük (4KB veya 8KB) sayfa boyutu kullanılır. Ancak, "akıllı" bir uygulama talep ederse, adres alanının belirli bir bölümü için tek bir büyük sayfa (örneğin, 4MB boyutunda) kullanılabilir. bu tür uygulamaların sık kullanılan (ve büyük) bir veri yapısını böyle bir alana yalnızca tek bir TLB girişi tüketirken yerleştirmesini sağlamak. Bu tür büyük sayfa kullanımı, veritabanı yönetim sistemlerinde ve diğer üst düzey ticari uygulamalarda yaygındır. Bununla birlikte, birden fazla sayfa boyutunun ana nedeni, sayfa tablosu alanından tasarruf etmek değildir; TLB üzerindeki baskıyı azaltmak, bir programın çok fazla TLB hatası yaşamadan adres alanının daha fazlasına erişmesini sağlamaktır. Bununla birlikte, araştırmacıların [N+02] gösterdiği gibi, birden fazla sayfa boyutu kullanmak işletim sistemi sanal bellek yöneticisini önemli ölçüde daha karmaşık hale getirir ve bu nedenle büyük sayfalar bazen en kolay şekilde, büyük sayfaları doğrudan talep etmek için uygulamalara yeni bir arayüz dışa aktararak kullanılır.

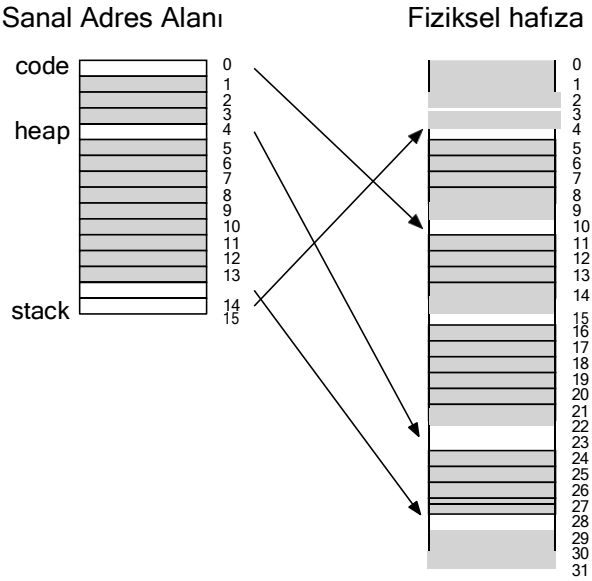
Sayfa tablosunun boyutunda dört azalma (şaşırtıcı olmayan bir şekilde, küçülme sayfa boyutundaki dört artış faktörünü tam olarak yansıtır).

Bununla birlikte, bu yaklaşımla ilgili en büyük sorun, büyük sayfaların her sayfada israfa yol açmasıdır; bu, **internal fragmentation (dahili parçalanma)** olarak bilinen bir sorundur (çünkü atık, **internal (tahsis biriminin)** içindedir). Böylece uygulamalar, sayfaları ayırmaya başlar, ancak her birinden yalnızca küçük parçalar kullanır ve bellek bu aşırı büyük sayfalarla hızla dolar. Bu nedenle, çoğu sistem genel durumda nispeten küçük sayfa boyutları kullanır: 4KB (x86'da olduğu gibi) veya 8KB (SPARCv9'da olduğu gibi). Sorunumuz bu kadar basit çözülmeyecek ne yazık ki.

20.2 Hibrit Yaklaşım: Sayfalama ve Segmentler

Hayatta bir şeye iki makul ama farklı yaklaşımınız olduğunda, her iki dünyanın da en iyisini elde edip edemeyeceğinizi görmek için her zaman ikisinin birleşimini incelemelisiniz. Böyle bir kombinasyona **hibrit(hybrid)** diyoruz. Örneğin, ikisini Reese's Fıstık Ezmesi Kupası [M28] olarak bilinen hoş bir melezde birleştirebilecekken neden sadece çikolata veya sade fıstık ezmesi yiyorsunuz?

Yıllar önce, Multics'in yaratıcıları (özellikle Jack Dennis), Multics sanal bellek sisteminin [M07] yapımında böyle bir fikre rastladılar. Özellikle Dennis, sayfa tablolarının bellek yükünü azaltmak için sayfalama ve bölümlemeyi birleştirme fikrine sahipti. Tipik bir doğrusal sayfa tablosunu daha ayrıntılı inceleyerek bunun neden işe yarayabileceğini görebiliriz. Yığın ve yığının kullanılan bölümlerinin küçük olduğu bir adres alanımız olduğunu varsayalım. Örnek olarak, 1KB sayfaları olan 16KB'lik küçük bir adres alanı kullanıyoruz (Şekil 20.1); bu adres alanı için sayfa tablosu Şekil 20.2'dedir.



Şekil 20.1: **1KB Sayfalı 16KB Adres Alanı**

PFN	geçerli	korum	mevcut	kirli
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Şekil 20.2: **16 KB Adres Alanı İçin Sayfa Tablosu**

Bu örnek, tek kod sayfasının (VPN 0) fiziksel sayfa 10 ile eşlendiğini varsayar, tek yığın sayfasından (VPN 4) fiziksel sayfa 23'eve adres alanının diğer ucundaki iki yığın sayfası (VPN 14 ve 5), sırasıyla 28 ve 4 numaralı fiziksel sayfalarla eşlenir.

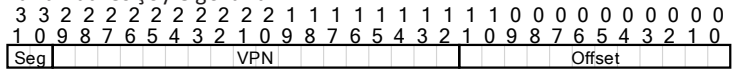
Resimden de görebileceğiniz gibi, sayfa tablosunun çoğu kullanılmamış, **geçersiz(invalid)** girişlerle dolu. Ne kadar da boş! Ve bu, 16 KB'lik küçük bir adres alanı içindir. 32 bitlik bir adres alanının sayfa tablosunu ve orada boşa harcanan tüm potansiyel alanı hayal edin! Aslında öyle bir şey hayal etmeyin; çok ürkütücü.

Bu nedenle hibrit yaklaşımımız: sürecin tüm adres alanı için tek sayfalık bir tabloya sahip olmak yerine, neden mantıksal segment başına bir tane olmasın? Bu örnekte, biri adres alanının kod, yığın ve yığın bölümleri için olmak üzere üç sayfa tablomuz olabilir.

Şimdi, segmentasyon ile hatırlayın, her bir segmentin fiziksel bellekte nerede yaşadığını ve bir **sınırı(bound)**veya **limiti(limit)**bize söyleyen bir temel kaydımız vardı. bize söz konusu segmentin boyutunu söyleyen kayıt**.** Hibridimizde, MMU'da hala bu yapılara sahibiz; burada tabanı segmentin kendisine işaret etmek için değil, o segmentin sayfa tablosunun fiziksel adresini tutmak için kullanıyoruz. Sınır kaydı, sayfa tablosunun sonunu belirtmek için kullanılır (yani, kaç tane geçerli sayfaya sahip olduğu).

Açıklığa kavuşturmak için basit bir örnek yapalım. 4 KB sayfaları olan 32 bitlik bir sanal adres alanı ve dört parçaya bölünmüş bir adres alanı varsayalım. Bu örnek için yalnızca üç segment kullanacağız: biri kod için, biri yığın için ve biri de yığın için.

Bir adresin hangi segmente ait olduğunu belirlemek için adres uzayının en üstteki iki bitini kullanacağız. Kod için 01, yığın için 10 ve yığın için 11 olmak üzere 00'ın kullanılmayan bölüm olduğunu varsayalım. Böylece, sanal bir adres şöyle görünür:



Donanımda, her biri kod, yığın ve yığın için birer tane olmak üzere üç temel/sınır çifti olduğunu varsayalım. Bir işlem çalışırken, bu bölümlerin her biri için temel kayıt, o bölüm için bir doğrusal sayfa tablosunun fiziksel adresini içerir; bu nedenle, sistemdeki her işlem artık kendisiyle ilişkilendirilmiş üç sayfa tablosuna sahiptir. Bağlam anahtarında, bu kayıtlar, yeni çalışan sürecin sayfa tablolarının konumunu yansıtacak şekilde değiştirilmelidir.

Bir TLB hatasında (donanım tarafından yönetilen bir TLB varsayıldığında, yani donanımın TLB kayıplarını işlemekten sorumlu olduğu durumda), donanım, hangi taban ve sınır çiftinin kullanılacağını belirlemek için segment bitlerini (SN) kullanır. Donanım daha sonra buradaki fiziksel adresi alır ve sayfa tablosu girişinin (PTE) adresini oluşturmak için aşağıdaki gibi VPN ile birleştirir:

```
SN           = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN          = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

Bu sıra tanıdık gelmeli; daha önce doğrusal sayfa tablolarında gördüğümüzle hemen hemen aynıdır. Tek fark, elbette, tek sayfalı tablo tabanlı kayıt yerine üç segmentli temel kayıtlardan birinin kullanılmasıdır.

İPUCU: HİBRİT KULLANIN

İki iyi ve görünüşte karşıt fikriniz olduğunda, her iki dünyanın da en iyisini elde etmeyi başaran bir **melezde(hybrid)** birleştirip birleştiremeyeceğinizi her zaman görmelisiniz. Örneğin hibrit mısır türlerinin, doğal olarak oluşan tüm türlerden daha sağlam olduğu bilinmektedir. Tabii ki, tüm melezler iyi bir fikir değildir; Zeedonk'a (veya Zonkey'e) bakın, Zebra ve Eşeğin melezidir. Böyle bir yaratığın var olduğuna inanmıyorsanız, bakın ve şaşırmaya hazırlanın.

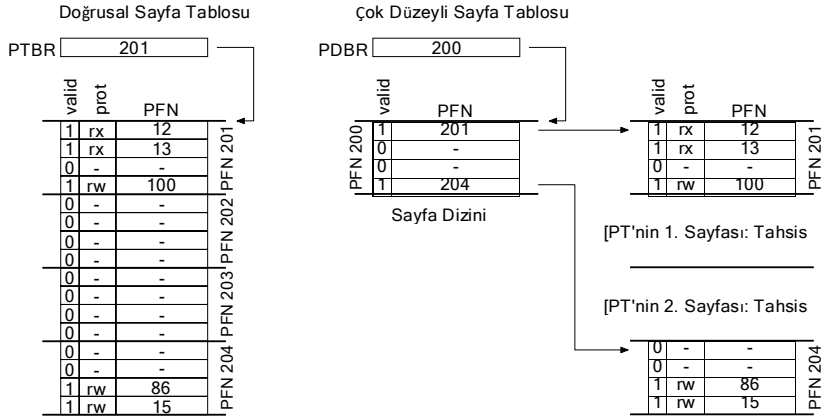
Hibrit şemamızdaki kritik fark, segment başına bir sınır kaydının varlığıdır; her sınır kaydı, segmentteki maksimum geçerli sayfanın değerini tutar. Örneğin, kod bölümü ilk üç sayfasını (0, 1 ve 2) kullanıyorsa, kod bölümü sayfa tablosunun kendisine tahsis edilmiş yalnızca üç girişi olacaktır ve sınır kaydı 3'e ayarlanacaktır; segmentin sonunun ötesindeki bellek erişimleri bir istisna oluşturacak ve muhtemelen sürecin sonlandırılmasına yol açacaktır. Bu şekilde hibrit yaklaşımımız, doğrusal sayfa tablosuna kıyasla önemli bir bellek tasarrufu gerçekleştirir; yığın ile yığın arasındaki ayrılmamış sayfalar artık bir sayfa tablosunda yer kaplamaz (yalnızca geçerli değil olarak işaretlemek için).

Ancak, fark edebileceğiniz gibi, bu yaklaşım sorunsuz değildir. Birincisi, hala segmentasyon kullanmamızı gerektiriyor; daha önce tartıştığımız gibi, adres alanının belirli bir kullanım modelini varsaydığından, segmentasyon istediğimiz kadar esnek değildir; örneğin, büyük ama seyrek kullanılan bir yığınımız varsa, yine de çok fazla sayfa tablosu israfı yaşayabiliriz. İkincisi, bu melez dış parçalanmanın yeniden ortaya çıkmasına neden olur. Belleğin çoğu sayfa boyutlu birimlerde yönetilirken, sayfa tabloları artık isteğe bağlı boyutta olabilir (PTE'lerin katları halinde). Bu nedenle, bellekte onlar için boş alan bulmak daha karmaşıktır. Bu nedenlerden dolayı, insanlar daha küçük sayfa tabloları uygulamak için daha iyi yollar aramaya devam ettiler.

20.3 Çok Düzeyli Sayfa Tabloları

Farklı bir yaklaşım, segmentasyona dayanmaz, ancak aynı soruna saldırır: hepsini hafızada tutmak yerine sayfa tablosundaki tüm bu geçersiz bölgelerden nasıl kurtuluruz? Doğrusal sayfa tablosunu ağaç gibi bir şeye dönüştürdüğü için bu yaklaşımı **çok düzeyli sayfa tablosu(multi-level page table)** olarak adlandırıyoruz. Bu yaklaşım o kadar etkilidir ki birçok modern sistem bunu kullanır (örneğin, x86 [BOH10]). Şimdi bu yaklaşımı ayrıntılı olarak açıklıyoruz.

Çok seviyeli bir sayfa tablosunun arkasındaki temel fikir basittir. İlk olarak, sayfa tablosunu sayfa boyutunda birimlere ayırın; daha sonra, sayfa tablosu girişlerinden (PTE'ler) oluşan bir sayfanın tamamı geçersizse, sayfa tablosunun o sayfasını hiç tahsis etmeyin. Sayfa tablosunun bir sayfasının geçerli olup olmadığını (ve geçerliyse bellekte nerede olduğunu) izlemek için **sayfa dizini(page directory)** adı verilen yeni bir yapı kullanın. Böylece sayfa dizini, size sayfa tablosunun bir sayfasının nerede olduğunu veya sayfa tablosunun tüm sayfasının geçerli sayfa içermediğini söylemek için kullanılabilir.



Şekil 20.3: **Doğrusal (Sol) ve Çok Düzeyli (Sağ) Sayfa Tabloları**

Şekil 20.3 bir örnek göstermektedir. Şeklin solunda klasik doğrusal sayfa tablosu bulunur; adres alanının orta bölgelerinin çoğu geçerli olmasa da, yine de bu bölgeler için ayrılmış sayfa tablosu alanına ihtiyacımız var (yani, sayfa tablosunun ortadaki iki sayfası). Sağda çok düzeyli bir sayfa tablosu var. Sayfa dizini, sayfa tablosunun yalnızca iki sayfasını geçerli olarak işaretler (ilk ve son); bu nedenle, sayfa tablosunun yalnızca bu iki sayfası bellekte bulunur. Ve böylece, çok düzeyli bir tablonun ne yaptığını görselleştirmenin bir yolunu görebilirsiniz: doğrusal sayfa tablosunun bazı kısımlarını ortadan kaldırır (bu çerçeveleri başka kullanımlar için serbest bırakır) ve sayfa tablosunun hangi sayfalarının sayfa dizini ile tahsis edildiğini izler.

İki seviyeli basit bir tablodaki sayfa dizini, sayfa tablosunun her sayfası için bir giriş içerir. Bir dizi **sayfa dizini girişinden(page directory entries) (PDE)** oluşur. Bir PDE (en azından), bir PTE'ye benzer şekilde, geçerli bir bit ve bir **sayfa çerçeve numarasına(page frame number) (PFN)** sahiptir. Ancak, yukarıda ima edildiği gibi, bu geçerli bitin anlamı biraz farklıdır: PDE geçerliyse, bu, sayfa tablosunun girişin işaret ettiği sayfalarından en az birinin (PFN aracılığıyla) geçerli olduğu anlamına gelir, yani bu PDE tarafından işaret edilen o sayfadaki en az bir PTE'de, geçerli bu PTE'deki bit bire ayarlanır. PDE geçerli değilse (yani sıfıra eşitse), PDE'nin geri kalanı tanımlanmamıştır.

Çok düzeyli sayfa tabloları, şimdiye kadar gördüğümüz yaklaşımlara göre bazı bariz avantajlara sahiptir. Birincisi ve belki de en bariz olanı, çok düzeyli tablo yalnızca kullandığınız adres alanı miktarıyla orantılı olarak sayfa tablosu alanı tahsis eder; bu nedenle genellikle kompakttır ve seyrek adres alanlarını destekler.

İkincisi, dikkatli bir şekilde oluşturulursa, sayfa tablosunun her bölümü bir sayfaya düzgün bir şekilde sığar ve belleği yönetmeyi kolaylaştırır; İşletim sistemi, bir sayfa tablosu ayırması veya büyütmesi gerektiğinde bir sonraki boş sayfayı alabilir.

İPUCU: ZAMAN-UZAY DENGELERİNİ ANLAYIN

Bir veri yapısı oluştururken, yapımında her zaman **zaman-uzay takasları(time-space trade-offs)** düşünülmelidir. Genellikle, belirli bir veri yapısına daha hızlı erişim sağlamak isterseniz, yapı için bir alan kullanım cezası ödemeniz gerekir.

Bunu, yalnızca VPN tarafından dizine eklenmiş bir PTE dizisi olan basit (disk belleği olmayan) bir doğrusal sayfa tablosu[2] ile karşılaştırın; böyle bir yapıyla, doğrusal sayfa tablosunun tamamı bitişik olarak fiziksel bellekte bulunmalıdır. Büyük bir sayfa tablosu için (diyelim ki 4 MB), bu kadar büyük bir kullanılmayan bitişik boş fiziksel bellek yığını bulmak oldukça zor olabilir. Çok düzeyli bir yapıyla, sayfa tablosunun parçalarına işaret eden sayfa dizini kullanılarak bir **dolaylı düzey(level of indirection)** ekliyoruz; bu dolaylı yol, sayfa tablosu sayfalarını fiziksel bellekte istediğimiz yere yerleştirmemizi sağlar.

Çok düzeyli tabloların bir maliyeti olduğu unutulmamalıdır; TLB hatasında, sayfa tablosundan doğru çeviri bilgisini almak için bellekten iki yükleme gerekir (biri sayfa dizini için, diğeri PTE'nin kendisi içindoğrusal bir sayfa tablosuna sahip tek bir yükün aksine. Bu nedenle, çok düzeyli tablo, zaman-uzay dengesine küçük bir örnektir. Daha küçük masalar istedik (ve aldık), ama bedava değil; Genel durumda (TLB vuruşu), performans açık bir şekilde aynı olsa da, bir TLB eksikliği bu daha küçük masa ile daha yüksek bir maliyetten muzdariptir.

Diğer bir belirgin olumsuzluk ise karmaşıklıktır. Sayfa tablosu aramasını yapan ister donanım ister işletim sistemi olsun (bir TLB hatasında), bunu yapmak şüphesiz basit bir doğrusal sayfa tablosu aramasından daha karmaşıktır. Genellikle performansı artırmak veya genel giderleri azaltmak için karmaşıklığı artırmaya istekliyiz; çok düzeyli bir tablo söz konusu olduğunda, değerli bellekten tasarruf etmek için sayfa tablosu aramalarını daha karmaşık hale getiririz.

Ayrıntılı Çok Düzeyli Bir Örnek

Çok düzeyli sayfa tablolarının ardındaki fikri daha iyi anlamak için bir örnek yapalım. 64 bayt sayfaları olan 16 KB boyutunda küçük bir adres alanı hayal edin. Böylece, VPN için 8 bit ve ofset için 6 bit olmak üzere 14 bitlik bir sanal adres alanımız var. Adres alanının yalnızca küçük bir kısmı kullanımda olsa bile, bir doğrusal sayfa tablosunda 2 (256) giriş olacaktır. Şekil 20.4 (sayfa 8), böyle bir adres alanının bir örneğini sunar.

Bu örnekte, sanal sayfalar 0 ve 1 kod içindir, sanal sayfalar 4 ve 5 öbek içindir ve sanal sayfalar 254 ve 255 yığın içindir; adres alanının geri kalan sayfaları kullanılmaz.

Bu adres alanı için iki seviyeli bir sayfa tablosu oluşturmak için, tam doğrusal sayfa tablomuzla başlayıp onu sayfa boyutlu birimlere ayırıyoruz. Tam tablomuzu hatırlayın (bu örnekte) 256 giriş var; her bir PTE'nin 4 bayt boyutunda olduğunu varsayalım.

[2] Burada bazı varsayımlarda bulunuyoruz, yani tüm sayfa tablolarının bütünüyle fiziksel bellekte bulunduğu (yani, diske takas edilmedikleri); yakında bu varsayımı gevşeteceğiz.

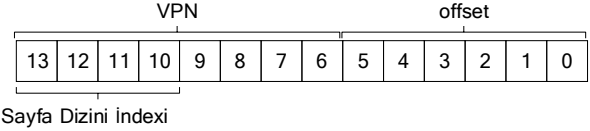
0000 0000	code
0000 0001	code
0000 0010	(free)
0000 0011	(free)
0000 0100	heap
0000 0101	heap
0000 0110	(free)
0000 0111	(free)
.....	... all free ...
1111 1100	(free)
1111 1101	(free)
1111 1110	stack
1111 1111	stack

Şekil 20.4: **64 bayt Sayfalı 16 KB Adres Alanı**

Böylece sayfa tablomuz 1KB (256×4 byte) boyutundadır. 64 baytlık sayfalarımız olduğu göz önüne alındığında, 1KB sayfa tablosu 16 adet 64 baytlık sayfaya bölünebilir; her sayfa 16 PTE tutabilir.

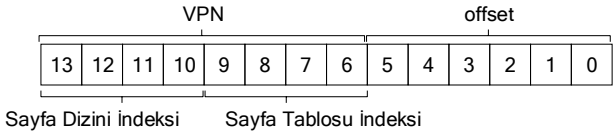
Şimdi anlamamız gereken şey, bir VPN'i nasıl alıp önce sayfa dizinine sonra da sayfa tablosunun sayfasına indekslemek için nasıl kullanacağımızdır. Her birinin bir giriş dizisi olduğunu unutmayın; bu nedenle, çözmemiz gereken tek şey, VPN parçalarından her biri için dizini nasıl oluşturacağımızdır.

Önce sayfa dizinine indeksleyelim. Bu örnekteki sayfa tablomuz küçüktür: 16 sayfaya yayılmış 256 giriş. Sayfa dizini, sayfa tablosunun her sayfası için bir girişe ihtiyaç duyar; dolayısıyla 16 girişi vardır. Sonuç olarak, dizine endekslemek için VPN'nin dört bitine ihtiyacımız var; VPN'nin ilk dört bitini şu şekilde kullanıyoruz:



Sayfa dizini dizinini(**page-directory index**) (kısaca PDIndex) VPN'den çıkardıktan sonra, basit bir hesaplamayla sayfa dizini girişinin (PDE) adresini bulmak için kullanabiliriz: PDEAddr = PageDirBase + (PDIndex * sizeof(PDE)). Bu, çevirimizde daha fazla ilerleme kaydetmek için şimdi incelediğimiz sayfa dizinimiz ile sonuçlanır.

Sayfa dizini girişi geçersiz olarak işaretlenirse, erişimin geçersiz olduğunu biliriz ve bu nedenle bir istisna oluştururuz. Bununla birlikte, PDE geçerliyse, yapacak daha çok işimiz var. Spesifik olarak, şimdi sayfa tablosu girişini (PTE), bu sayfa dizini girişi tarafından işaret edilen sayfa tablosu sayfasından getirmemiz gerekiyor. Bu PTE'yi bulmak için, VPN'nin kalan bitlerini kullanarak sayfa tablosunun bölümüne indekslemeliyiz:



Bu **sayfa tablosu dizini(page-table index)** (kısaca PTIndex), daha sonra sayfa tablosunun kendisini dizinlemek için kullanılabilir ve bize PTE'mizin adresini verir:

```
PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
```

Sayfa dizini girişinden elde edilen sayfa çerçevesi numarasının (PFN), PTE'nin adresini oluşturmak için sayfa tablosu dizini ile birleştirilmeden önce sola kaydırılması gerektiğine dikkat edin.

Tüm bunların mantıklı olup olmadığını görmek için şimdi çok düzeyli bir sayfa tablosunu bazı gerçek değerlerle dolduracağız ve tek bir sanal adresi çevireceğiz. Bu örnek için **sayfa dizini(page directory)** ile başlayalım (Şekil 20.5'in sol tarafı).

Şekilde, her sayfa dizini girişinin (PDE), adres alanı için sayfa tablosunun bir sayfası hakkında bir şeyler açıkladığını görebilirsiniz. Bu örnekte, adres alanında (başlangıçta ve sonda) iki geçerli bölgemiz ve arada bir dizi geçersiz eşlememiz var.

Fiziksel sayfa 100'de (sayfa tablosunun 0. sayfasının fiziksel çerçeve numarası), adres alanındaki ilk 16 VPN için 16 sayfa tablo girişinin ilk sayfasına sahibiz. Sayfa tablosunun bu bölümünün içeriği için bkz. Şekil 20.5 (orta kısım).

Sayfa tablosunun bu sayfası, ilk 16 VPN için eşlemeleri içerir; Örneğimizde, VPN'ler 0 ve 1 geçerlidir (kod bölümü), 4 ve 5 de (yığın).

Sayfa Dizini		**PT Sayfası (@PFN:100)**			**PT Sayfası (@PFN:101)**		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

Şekil 20.5: **Bir Sayfa Dizini ve Sayfa Tablosu Parçaları**

İPUCU: KARMAŞIKLIĞA DİKKAT EDİN

Sistem tasarımcıları, sistemlerine karmaşıklık eklemek konusunda dikkatli olmalıdır. İyi bir sistem kurucunun yaptığı, eldeki görevi yerine getiren en az karmaşık sistemi uygulamaktır. Örneğin, disk alanı bolsa, mümkün olduğunca az bayt kullanmak için çok çalışan bir dosya sistemi tasarlamamalısınız; benzer şekilde, işlemciler hızlıysa, işletim sistemi içinde temiz ve anlaşılır bir modül yazmak, eldeki görev için belki de CPU için en iyi duruma getirilmiş, elle birleştirilmiş kod yazmaktan daha iyidir. Zamanından önce optimize edilmiş kod veya diğer biçimlerdeki gereksiz karmaşıklığa karşı dikkatli olun; bu tür yaklaşımlar sistemlerin anlaşılmasını, bakımını ve hata ayıklamasını zorlaştırır. Antoine de Saint-Exupery'nin meşhur yazdığı gibi: "Mükemmelliğe, eklenecek bir şey kalmadığında değil, çıkarılacak bir şey kalmadığında ulaşılır." Yazmadığı şey: "Mükemmellik hakkında bir şeyler söylemek, onu gerçekten başarmaktan çok daha kolay."

Böylece tablo, bu sayfaların her biri için eşleme bilgisine sahiptir. Girişlerin geri kalanı geçersiz olarak işaretlendi.

Sayfa tablosunun diğer geçerli sayfası PFN 101 içinde bulunur. Bu sayfa, adres alanının son 16 VPN'i için eşlemeler içerir; ayrıntılar için bkz. Şekil 20.5 (sağ).

Örnekte, VPN 254 ve 255 (yığın) geçerli eşlemelere sahiptir. Umarız, bu örnekte görebildiğimiz şey, çok düzeyli dizinlenmiş bir yapıyla ne kadar alan tasarrufunun mümkün olduğudur. Bu örnekte, on altı sayfanın tamamını bir doğrusal sayfa tablosuna ayırmak yerine yalnızca üç sayfa ayırdık: biri sayfa dizini için ve ikisi sayfa tablosunun geçerli eşlemelere sahip parçaları için. Büyük (32-bit veya 64-bit) adres alanlarından elde edilen tasarruf açıkça çok daha fazla olabilir.

Son olarak, bir çeviri yapmak için bu bilgileri kullanalım. İşte VPN 254'ün 0. baytına atıfta bulunan bir adres: 0x3F80 veya ikili olarak 11 1111 1000 0000.

Sayfa dizinine indekslemek için VPN'nin ilk 4 bitini kullanacağımızı hatırlayın. Böylece, 1111, yukarıdaki sayfa dizininin son (0'dan başlarsanız 15'inci) girişini seçecektir. Bu bizi, adres 101'de bulunan sayfa tablosunun geçerli bir sayfasına yönlendirir. Ardından, sayfa tablosunun o sayfasına endekslemek ve istenen PTE'yi bulmak için VPN'nin (1110) sonraki 4 bitini kullanırız. 1110, sayfadaki sondan sonraki (14.) giriştir ve bize sanal adres alanımızın 254. sayfasının fiziksel sayfa 55'te eşlendiğini söyler. PFN=55 (veya hex 0x37)'yi offset=000000 ile birleştirerek, böylece istediğimiz fiziksel adresi oluşturabilir ve talebi bellek sistemine gönderebiliriz:

```
PhysAddr = (PTE.PFN << SHIFT) + offset
= 00 1101 1100 0000 = 0x0DC0.
```

Artık, sayfa tablosunun sayfalarına işaret eden bir sayfa dizini kullanarak iki düzeyli bir sayfa tablosunun nasıl oluşturulacağı hakkında bir fikriniz olmalıdır. Ancak maalesef işimiz bitmedi. Şimdi tartışacağımız gibi, bazen iki seviyeli sayfa tablosu yeterli olmaz!

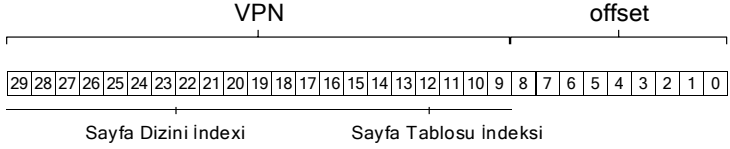
İkiden Fazla Düzey

Şimdiye kadarki örneğimizde, çok düzeyli sayfa tablolarının yalnızca iki düzeyi olduğunu varsaydık: bir sayfa dizini ve ardından sayfa tablosunun parçaları. Bazı durumlarda, daha derin bir ağaç mümkündür (ve aslında gereklidir).

Basit bir örnek alıp daha derin bir çok düzeyli tablonun neden yararlı olabileceğini göstermek için kullanalım. Bu örnekte, 30 bit sanal adres alanımız ve küçük (512 bayt) bir sayfamız olduğunu varsayalım. Böylece sanal adresimiz 21 bitlik bir sanal sayfa numarası bileşenine ve 9 bitlik bir kaymaya sahiptir.

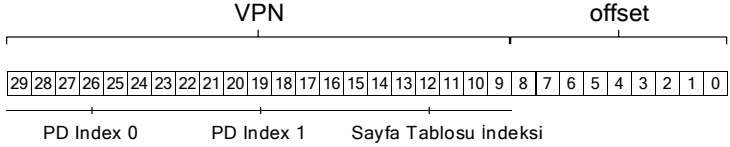
Çok düzeyli bir sayfa tablosu oluşturmadaki amacımızı unutmayın: sayfa tablosunun her bir parçasını tek bir sayfaya sığdırmak. Şimdiye kadar sadece sayfa tablosunun kendisini inceledik; ancak, sayfa dizini çok büyürse ne olur?

Sayfa tablosunun tüm parçalarını bir sayfaya sığdırmak için çok düzeyli bir tabloda kaç düzeyin gerekli olduğunu belirlemek için, bir sayfaya kaç sayfa tablosu girişinin sığacağını belirleyerek başlıyoruz. 512 bayt sayfa boyutumuz göz önüne alındığında ve PTE boyutunun 4 bayt olduğunu varsayarsak, tek bir sayfaya 128 PTE sığdırabileceğinizi görmelisiniz. Sayfa tablosunun bir sayfasını indekslediğimizde, indeks olarak VPN'nin en önemsiz 7 bitine (log2128) ihtiyacımız olacağı sonucuna varabiliriz:



Yukarıdaki diyagramdan da fark edebileceğiniz bir şey (büyük) sayfa dizininde kaç bit kaldığıdır: 14. Sayfa dizinimizin 2 girişi varsa, bir değil 128 sayfaya yayılıyor ve bu nedenle çok düzeyli sayfa tablosunun her parçasını bir sayfaya sığdırma hedefimiz kayboluyor.

Bu sorunu çözmek için, sayfa dizininin kendisini birden fazla sayfaya bölerek ve ardından sayfa dizininin sayfalarını işaret etmek için bunun üstüne başka bir sayfa dizini ekleyerek ağacın daha ileri bir seviyesini oluşturuyoruz. Böylece sanal adresimizi aşağıdaki gibi bölebiliriz:



Şimdi, üst düzey sayfa dizini dizini oluştururken, sanal adresin en üstteki bitlerini kullanıyoruz (şemada PD Dizini 0); bu dizin, sayfa dizini girişini üst düzey sayfa dizininden almak için kullanılabilir. Geçerliyse, sayfa dizininin ikinci düzeyine, üst düzey PDE'den fiziksel çerçeve numarası ve VPN'nin sonraki bölümü (PD Dizini 1) birleştirilerek başvurulur.

```
1  VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2  (Success, TlbEntry) = TLB_Lookup(VPN)
3  if (Success == True)   // TLB Hit
4    if (CanAccess(TlbEntry.ProtectBits) == True)
5      Offset   = VirtualAddress & OFFSET_MASK
6      PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
7      Register = AccessMemory(PhysAddr)
8    else
9      RaiseException(PROTECTION_FAULT)
10 else                  // TLB Miss
11   // first, get page directory entry
12   PDIndex = (VPN & PD_MASK) >> PD_SHIFT
13   PDEAddr = PDBR + (PDIndex * sizeof(PDE))
14   PDE     = AccessMemory(PDEAddr)
15   if (PDE.Valid == False)
16     RaiseException(SEGMENTATION_FAULT)
17   else
18     // PDE is valid: now fetch PTE from page table
19     PTIndex = (VPN & PT_MASK) >> PT_SHIFT
20     PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
21     PTE     = AccessMemory(PTEAddr)
22     if (PTE.Valid == False)
23       RaiseException(SEGMENTATION_FAULT)
24     else if (CanAccess(PTE.ProtectBits) == False)
25       RaiseException(PROTECTION_FAULT)
26     else
27       TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
28       RetryInstruction()
```

Şekil 20.6: **Çok Düzeyli Sayfa Tablosu Kontrol Akışı**

Son olarak, eğer geçerliyse, PTE adresi, ikinci düzey PDE'den gelen adresle birleştirilmiş sayfa tablosu dizini kullanılarak oluşturulabilir. vay! Bu çok iş. Ve hepsi sadece çok seviyeli bir tabloda bir şeye bakmak için.

Çeviri Süreci: TLB'yi unutmayın

İki seviyeli bir sayfa tablosu kullanarak tüm adres dönüştürme sürecini özetlemek için, kontrol akışını bir kez daha algoritmik biçimde sunuyoruz (Şekil 20.6). Şekil, her bellek referansında donanımda (donanım tarafından yönetilen bir TLB varsayılarak) ne olduğunu gösterir.

Şekilden de görebileceğiniz gibi, herhangi bir karmaşık çok düzeyli sayfa tablosu erişimi gerçekleşmeden önce, donanım önce TLB'yi kontrol eder; bir isabet üzerine, fiziksel adres, daha önce olduğu gibi, sayfa tablosuna hiç erişilmeden doğrudan oluşturulur. Donanımın tam çok düzeyli aramayı gerçekleştirmesi yalnızca bir TLB hatası durumunda gerekir. Bu yolda, geleneksel iki düzeyli sayfa tablomuzun maliyetini görebilirsiniz: geçerli bir çeviriyi aramak için iki ek bellek erişimi.

20.4 Ters Sayfa Tabloları

Tersine çevrilmiş sayfa tabloları(inverted page tables), sayfa tabloları dünyasında daha da fazla yer tasarrufu sağlar. Burada, birçok sayfa tablosuna (sistemin her işlemi için bir tane) sahip olmak yerine, sistemin her fiziksel sayfası için bir girişi olan tek bir sayfa tablosu tutuyoruz. Giriş, bize bu sayfayı hangi işlemin kullandığını ve bu işlemin hangi sanal sayfasının bu fiziksel sayfayla eşleştiğini söyler.

Doğru girişi bulmak, artık bu veri yapısında arama yapmak meselesidir. Doğrusal bir tarama pahalı olacaktır ve bu nedenle, aramaları hızlandırmak için genellikle temel yapı üzerine bir karma tablo oluşturulur. PowerPC, böyle bir mimarinin bir örneğidir [JM98].

Daha genel olarak, tersine çevrilmiş sayfa tabloları, en başından beri söylediğimizi gösterir: sayfa tabloları yalnızca veri yapılarıdırVeri yapılarıyla onları daha küçük veya daha büyük, daha yavaş veya daha hızlı hale getirerek pek çok çılgınca şey yapabilirsiniz. Çok düzeyli ve ters çevrilmiş sayfa tabloları, yapılabilecek pek çok şeyin yalnızca iki örneğidir.

20.5 Sayfa Tablolarını Diske Dönüştürme

Son olarak, son bir varsayımın gevşetilmesini tartışıyoruz. Şimdiye kadar, sayfa tablolarının çekirdeğe ait fiziksel bellekte bulunduğunu varsaydıkSayfa tablolarının boyutunu küçültmek için yaptığımız pek çok hileye rağmen, bunların bir kerede belleğe sığmayacak kadar büyük olmaları yine de mümkündür. Bu nedenle, bazı sistemler bu tür sayfa tablolarını **çekirdek sanal belleğine(kernel virtual memory)** yerleştirerek, bellek baskısı biraz daraldığında sistemin bu sayfa tablolarından bazılarını diske **değiştirmesine(swap)** izin verir. Sayfaların bellek içinde ve bellek dışına nasıl taşınacağını daha ayrıntılı olarak anladığımızda, bundan sonraki bir bölümde (yani, VAX/VMS ile ilgili örnek olay incelemesinde) daha fazla konuşacağız.

20.6 Özet

Artık gerçek sayfa tablolarının nasıl oluşturulduğunu gördük; sadece doğrusal diziler olarak değil, daha karmaşık veri yapıları olarak. Bu tür tabloların sunduğu değiş tokuşlar zaman ve mekandadır -- tablo ne kadar büyükse, bir TLB'nin atlanması o kadar hızlı servis edilebilir ve bunun tersi de geçerlidir -- ve bu nedenle doğru yapı seçimi, büyük ölçüde verilen ortamın kısıtlamalarına bağlıdır.

Belleği kısıtlı bir sistemde (birçok eski sistem gibi), küçük yapılar mantıklıdır; makul miktarda belleğe ve aktif olarak çok sayıda sayfa kullanan iş yüklerine sahip bir sistemdeYazılım tarafından yönetilen TLB'lerde, veri yapılarının tüm alanı, işletim sistemi mucidinin zevkine açılır (ipucu: bu sizsiniz). Hangi yeni yapıları ortaya çıkarabilirsiniz? Hangi sorunları çözüyorlar? Uyurken bu soruları düşünün ve yalnızca işletim sistemi geliştiricilerinin görebileceği büyük hayaller kurun.

Referanslar

[BOH10] "Computer Systems: A Programmer's Perspective" by Randal E. Bryant and David R. O'Hallaron. Addison-Wesley, 2010. *We have yet to find a good first reference to the multi-level page table. However, this great textbook by Bryant and O'Hallaron dives into the details of x86, which at least is an early system that used such structures. It's also just a great book to have.*

[JM98] "Virtual Memory: Issues of Implementation" by Bruce Jacob, Trevor Mudge. IEEE Computer, June 1998. *An excellent survey of a number of different systems and their approach to virtualizing memory. Plenty of details on x86, PowerPC, MIPS, and other architectures.*

[LL82] "Virtual Memory Management in the VAX/VMS Operating System" by Hank Levy, P. Lipman. IEEE Computer, Vol. 15, No. 3, March 1982. *A terrific paper about a real virtual memory manager in a classic operating system, VMS. So terrific, in fact, that we'll use it to review everything we've learned about virtual memory thus far a few chapters from now.*

[M28] "Reese's Peanut Butter Cups" by Mars Candy Corporation. Published at stores near you. *Apparently these fine confections were invented in 1928 by Harry Burnett Reese, a former dairy farmer and shipping foreman for one Milton S. Hershey. At least, that is what it says on Wikipedia. If true, Hershey and Reese probably hate each other's guts, as any two chocolate barons should.*

[N+02] "Practical, Transparent Operating System Support for Superpages" by Juan Navarro, Sitaram Iyer, Peter Druschel, Alan Cox. OSDI '02, Boston, Massachusetts, October 2002. *A nice paper showing all the details you have to get right to incorporate large pages, or **superpages**, into a modern OS. Not as easy as you might think, alas.*

[M07] "Multics: History" Available: `http://www.multicians.org/history.html`. *This amazing web site provides a huge amount of history on the Multics system, certainly one of the most influential systems in OS history. The quote from therein: "Jack Dennis of MIT contributed influential architectural ideas to the beginning of Multics, especially the idea of combining paging and segmentation." (from Section 1.2.1)*

Ödev (Simülasyon)

Bu eğlenceli küçük ev ödevi, çok düzeyli bir sayfa tablosunun nasıl çalıştığını anlayıp anlamadığınızı test eder. Ve evet, bir önceki cümlede "eğlence" teriminin kullanımıyla ilgili bazı tartışmalar var. Programın adı, belki de şaşırtıcı olmayan bir şekilde: `paging-multilevel-translate.py` ayrıntılar için BENİOKU'ya bakın.

Sorular

1. Doğrusal bir sayfa tablosuyla, sayfa tablosunu bulmak için tek bir kayda ihtiyacınız vardır, donanımın bir TLB eksikliğinde arama yaptığını varsayarsak. İki seviyeli bir sayfa tablosunu bulmak için kaç kayda ihtiyacınız var? Üç seviyeli bir tablo mu?

Tek seviyeli bir sayfa tablosu için tek bir kayıt yeterlidir, ancak iki seviyeli bir sayfa tablosu için iki kayıt ve üç seviyeli bir sayfa tablosu için üç kayıt gerekir. Bu kayıtlar, bellekte bulunan sayfa tablosunun içeriğini tarama ve istenen sayfa için uygun bir adresi bulma işlemini gerçekleştirir. Bir TLB eksikliği durumunda, bu kayıtların sayısı önemli bir rol oynayabilir, çünkü bellekte bulunan sayfa tablosu taraması daha yavaş olacaktır.

2. Rastgele tohumlar 0, 1 ve 2 verilen çevirileri gerçekleştirmek için simülatörü kullanın ve -c işaretini kullanarak yanıtlarınızı kontrol edin. Her aramayı gerçekleştirmek için kaç tane bellek referansı gerekir?

./python3 paging-multilevel-translate.py

Bu süreci çalıştırdığımızda aşağıdaki gibi bir ekran görüntüsü oluşmaktadır

```
PDBR: 108  (decimal) [This means the page directory is held in this page]

Virtual Address 611c: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 3da8: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 17f5: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 7f6c: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 0bad: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 6d60: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 2a5b: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 4c5e: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 2592: Translates To What Physical Address (And Fetches what Value)? Or Fault?
Virtual Address 3e99: Translates To What Physical Address (And Fetches what Value)? Or Fault?
```

.

/python3 paging-multilevel-translate.py –s 1 –n 1 –c

Bu süreci çalıştırdığımızda aşağıdaki gibi bir ekran görüntüsü oluşmaktadır

```
PDBR: 17  (decimal) [This means the page directory is held in this page]

Virtual Address 0x6c74:
  --> pde index:0x1b [decimal 27] pde contents:0xa0 (valid 1, pfn 0x20 [decimal 32])
    --> pte index:0x3 [decimal 3] pte contents:0xe1 (valid 1, pfn 0x61 [decimal 97])
      --> Translates to Physical Address 0xc34 --> Value: 0x06
```

./python3 paging-multilevel-translate.py –s 0 –n 5 –c

Bu süreci çalıştırdığımızda aşağıdaki gibi bir ekran görüntüsü oluşmaktadır

```
PDBR: 108  (decimal) [This means the page directory is held in this page]

Virtual Address 0x611c:
  --> pde index:0x18 [decimal 24] pde contents:0xa1 (valid 1, pfn 0x21 [decimal 33])
    --> pte index:0x8 [decimal 8] pte contents:0xb5 (valid 1, pfn 0x35 [decimal 53])
      --> Translates to Physical Address 0x6bc --> Value: 0x08
Virtual Address 0x3da8:
  --> pde index:0xf [decimal 15] pde contents:0xd6 (valid 1, pfn 0x56 [decimal 86])
    --> pte index:0xd [decimal 13] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 0x17f5:
  --> pde index:0x5 [decimal 5] pde contents:0xd4 (valid 1, pfn 0x54 [decimal 84])
    --> pte index:0x1f [decimal 31] pte contents:0xce (valid 1, pfn 0x4e [decimal 78])
      --> Translates to Physical Address 0x9d5 --> Value: 0x1c
Virtual Address 0x7f6c:
  --> pde index:0x1f [decimal 31] pde contents:0xff (valid 1, pfn 0x7f [decimal 127])
    --> pte index:0x1b [decimal 27] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
      --> Fault (page table entry not valid)
Virtual Address 0x0bad:
  --> pde index:0x2 [decimal 2] pde contents:0xe0 (valid 1, pfn 0x60 [decimal 96])
    --> pte index:0x1d [decimal 29] pte contents:0x7f (valid 0, pfn 0x7f [decimal 127])
      --> Fault (page table entry not valid)

kadir@kadir-virtual-machine:~/Desktop/ostep/ostep-homework/vm-smalltables$
```

3. Önbelleğin nasıl çalıştığını anladığınıza göre, sayfa tablosuna yapılan bellek referanslarının önbellekte nasıl davranacağını düşünüyorsunuz? Çok sayıda önbellek isabetine (ve dolayısıyla hızlı erişime mi) yol açacaklar yoksa çok sayıda ıskalamaya (ve dolayısıyla yavaş erişime) mi yol açacaklar?

Önbellek referansları, bellekte saklanan verilerin hızlı erişimini sağlamak amacıyla kullanılır. Önbellekte tutulan veriler, işlemcinin çalıştığı işlemlerin çoğunu hızlandıracak ve dolayısıyla sayfa tablosunda yapılan bellek referanslarının çok sayıda önbellek isabetine yol açması beklenir. Ancak, eğer önbellekte saklanan veriler işlemcinin ihtiyaç duyduğu veriler değilse, bu verilerin ıskalanması ve dolayısıyla yavaş erişim sonucu ortaya çıkabilir. Bu nedenle, önbellekte saklanan verilerin işlemcinin ihtiyaç duyabileceği veriler olması önemlidir.