20

Paging: Smaller Tables (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. Hatırlayacağınız gibi doğrusal sayfa tabloları oldukça büyür.4 KB (2^{12} bayt) sayfa ve 4 bayt sayfa tablosu girişi ile yeniden 32 bitlik bir adres alanı (2^{32} bayt) varsayalım. Dolayısıyla, bir adres alanı içinde kabaca bir milyon sanal sayfa bulunur ($2^{32}:2^{12}$) ; 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 nadir değildir), yalnızca sayfa tabloları için yüzlerce megabayt bellek ayıracağız! Sonuç olarak bu ağır yükü azaltmak için bazı teknikler arayışındayız. Bu tekniklerin de birçoğu var, hadi başlayalım. Ama bizim püf noktamızdan önce değil:

CRUX: HOW TO MAKE PAGE TABLES SMALLER?(PÜF NOKTASI: SAYFA TABLOLARI NASIL DAHA KÜÇÜK YAPILIR?)

Basit dizi tabanlı sayfa tabloları (genellikle doğrusal sayfa tabloları olarak adlandırılır) çok büyüktür ve tipik sistemlerde çok fazla bellek kaplar. Sayfa tablolarını nasıl küçültebiliriz? Ana fikirler nelerdir? Bu yeni veri yapılarının bir sonucu olarak hangi verimsizlikler ortaya çıkıyor?

20.1 Simple Solution: Bigger Pages(Basit Çözüm: Daha Büyük Sayfalar)

Sayfa tablosunun boyutunu tek bir basit yolla yani daha büyük sayfalar kullanarak azaltabiliriz. 32 bit adres alanımızı tekrar alalım, ancak bu sefer 16 KB sayfaları varsayalım. Böylece 18 bitlik bir VPN artı 14 bitlik bir kaymamız olur. Her bir PTE için aynı boyutu (4 bayt) topladığımızda, artık lineer sayfa tablomuzda 2^{18} girdimiz oluşur ve dolayısıyla sayfa tablosu başına toplam 1 MB boyut, Ya da aslında, yapmayabilirsin; Bu çağrı olayı kontrolden çıkıyor, değil mi? Bununla birlikte, çözüme geçmeden önce çözdüğünüz problemi anladığınızdan her zaman emin olun; aslında, eğer problemi anlarsanız, genellikle çözümü kendiniz de elde edebilirsiniz. Burada problem açık olmalı: basit doğrusal (dizi tabanlı) sayfa tabloları çok büyük.

ASIDE: MULTIPLE PAGE SIZES(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. Bununla birlikte, "akıllı" bir uygulama talep ederse, adres alanının belirli bir bölümü için tek bir büyük sayfa (örneğin, 4MB boyutunda) kullanılabilir ve bu tür uygulamaların, yalnızca tek bir TLB girişi tüketirken, sık kullanılan (ve büyük) bir veri yapısını böyle bir alana yerleştirmesine olanak tanır. 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 daha fazla adres alanına 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 büyük sayfaları doğrudan talep etmek için uygulamalara yeni bir arayüz dışa aktararak en kolay şekilde kullanılır.

sayfa tablosunun boyutunda dört kat azalma olur (şaşırtıcı olmayan bir şekilde, küçültme sayfa boyutundaki dört artış faktörünü tam olarak yansıtır).

Bununla birlikte, bu yaklaşımla ilgili en büyük sorun, büyük sayfaların her sayfada israfa yol açmasıdır; **internal fragmentation (iç parçalanma)** olarak bilinen bir sorundur (çünkü atık, tahsis birimine **internal (dahil)** dir. 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). Ne yazık ki sorunumuz bu kadar basit çözülmeyecek.

20.2 Hybrid Approach: Paging and Segments (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 **invalid (kombinasyon)** hibrit diyoruz. Örneğin, ikisini Reese's Fıstık Ezmesi Kupası (the Reese's Peanut Butter Cup) [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.

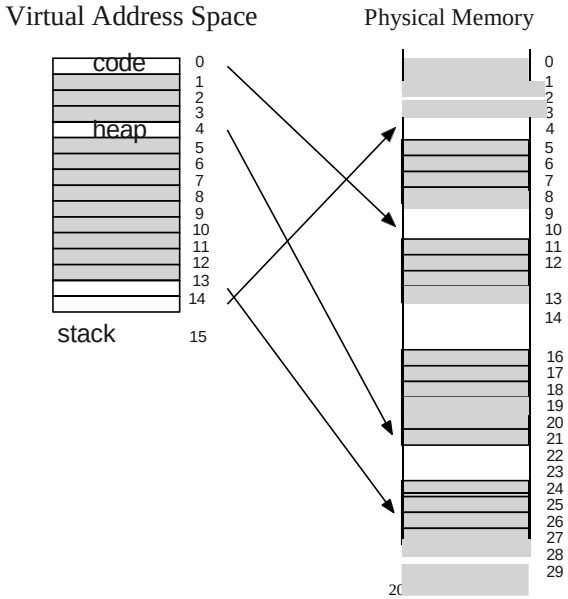


Figure 20.1: **A 16KB Address Space With 1KB Pages**

PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1

Figure 20.2: **A Page Table For 16KB Address Space**

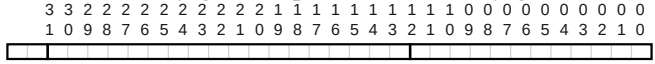
Resimden de görebileceğiniz gibi, sayfa tablosunun çoğu kullanılmamış, **invalid (geçersiz girişlerle)** dolu. Ne kadar da israf! Ve bu, 16 KB'lik küçük

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 şöyledir: sürecin tüm adres alanı için tek bir sayfa tablosuna sahip olmak yerine, neden mantıksal segment başına bir tane olmasın? Bu örnekte, biri adres alanının kod (code), yığın (heap) ve yığın (stack) bölümleri için olmak üzere üç sayfa tablomuz olabilir.

Şimdi, segmentasyonla hatırlayın, her bir segmentin fiziksel bellekte nerede yaşadığını söyleyen bir **base (temel)** kaydımız ve söz konusu segmentin boyutunu bize söyleyen bir **bound (sınır)** veya **limit (limit)** kaydımız vardı. 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 (code) için, biri yığın (heap) için ve biri de yığın (stack) için.



Donanımda, her biri **kod (code), yığın (heap) ve yığın (stack)** için birer tane olmak üzere üç temel/sınır çifti olduğunu varsayalım. Bir işlem çalışırken, bu kesimlerin 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 işlemin sayfa tablolarının konumunu yansıtacak şekilde değiştirilmelidir.

Bir **TLB (Etkin sayfalar ön belleği)** hatasında (donanım tarafından yönetilen bir TLB varsayıldığında, yani donanımın TLB kayıplarını işlemekten sorumlu olduğu durumda), donanım, hangi temel 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.

TIP: USE HYBRIDS

İki iyi ve görünüşte karşıt fikriniz olduğunda, her iki dünyanın da en iyisini elde etmeyi başaran bir hybrit (melez)de 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 (zebra) ve Eşeğin (donkey) melezidir.

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 işlemin sonlandırılmasına yol açacaktır. Bu şekilde hibrit yaklaşımımız, doğrusal sayfa tablosuna kıyasla önemli bir bellek tasarrufu gerçekleştirir; yığın (stack) ile yığın (heap) 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, segmentasyon, adres alanının belirli

bir kullanım modelini varsaydığından, 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.

20.3Multi-level Page Tables (Çok Düzeyli Sayfa Tabloları)

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

Multi-level Page Tables (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 ayırmayın. Sayfa tablosunun bir sayfasının geçerli olup olmadığını (ve geçerliyse bellekte nerede olduğunu) izlemek için **page directory (sayfa yöneticisi)** adı verilen yeni bir yapı kullanın. Böylece **page directory (sayfa yöneticisi)**, size sayfa tablosunun bir sayfasının nerede olduğunu veya sayfa tablosunun tüm sayfasının geçerli sayfa içermediğini söylemek için kullanılabilir.

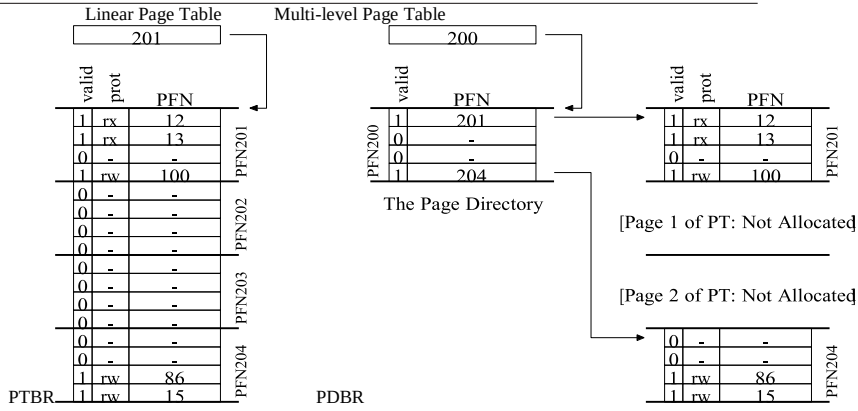


Figure 20.3: **Linear (Left) And Multi-Level (Right) Page Tables (Doğrusal (Sol) ve Çok Düzeyli (Sağ) Sayfa Tabloları)**

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 **Multi-level Page Tables (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, **Multi-level Page Tables (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 **page directory (sayfa yöneticisi)**, sayfa tablosunun her sayfası için bir giriş içerir. **Page directory entries** (**PDE**) **(Bir dizi sayfa dizini girişi) (PDE)** oluşur. Bir PDE (en azından), bir PTE'ye benzer şekilde, **valid bit (geçerli bit)** ve **page frame** number (sayfa çerçeve numarasına) (PFN) sahiptir. Bununla birlikte, yukarıda ima edildiği gibi, bu geçerli bitin anlamı biraz farklıdır: PDE geçerliyse, bu, girişin işaret ettiği (PFN yoluyla) sayfa tablosunun sayfalarından en az birinin geçerli olduğu anlamına gelir; yani , bu PDE tarafından işaret edilen sayfadaki en az bir PTE'de, o PTE'deki geçerli bit bir olarak ayarlanır. PDE geçerli değilse (yani sıfıra eşitse), PDE'nin geri kalanı tanımlanmamıştır.

Multi-level page tables (Çok düzeyli sayfa tabloları), şimdiye kadar gördüğümüz yaklaşımlara göre bazı bariz avantajlara sahiptir. Birincisi ve belki de en bariz olanı, **Multi-level Page Tables (çok düzeyli)**tablo yalnızca kullandığınız adres alanı miktarıyla orantılı olarak sayfa tablosu alanı ayırır; bu nedenle genellikle kompakttır ve seyrek reklam 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.

TIP: UNDERSTAND TIME-SPACE TRADE-OFFS

Bir veri yapısı oluşturulurken, yapımında her zaman **time-space trade-offs (zaman uzay takasları)** 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) doğrusal sayfa tablosu2 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. **Multi-level tables (Çok düzeyli bir yapıyla)**, sayfa tablosunun parçalarına işaret eden sayfa dizini kullanılarak bir **indirection (dolaylı düzey)** ekliyoruz; bu dolaylı yol, sayfa tablosu sayfalarını fiziksel bellekte istediğimiz yere yerleştirmemizi sağlar.

Multi-level tables (Çok düzeyli tabloların) bir maliyeti olduğu unutulmamalıdır; TLB hatasında, doğrusal sayfa tablosuyla yalnızca bir yükün aksine, sayfa tablosundan doğru çeviri bilgisini almak için (biri sayfa dizini için ve biri PTE'nin kendisi için) bellekten iki yükleme gerekli olacaktır. Bu nedenle, **Multi-level tables (Çok düzeyli tabloların)**, **time-space trade-off** (**zaman-uzay dengesine**) küçük bir örnektir. Daha küçük tablolar istedik (ve aldık), ama bedava değil; Genel durumda (TLB vuruşu), performans açık bir şekilde aynı olsa da, bir TLB eksikliği bu daha küçük tablo 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.

A Detailed Multi-Level Example (Ayrıntılı Çok Düzeyli Bir Örnek)

Multi-level tables (Çok düzeyli tabloların), 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 2^8 (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 (heap) içindir ve sanal sayfalar 254 ve 255 yığın (stack) içindir; adres alanının geri kalan sayfaları kullanılmaz.

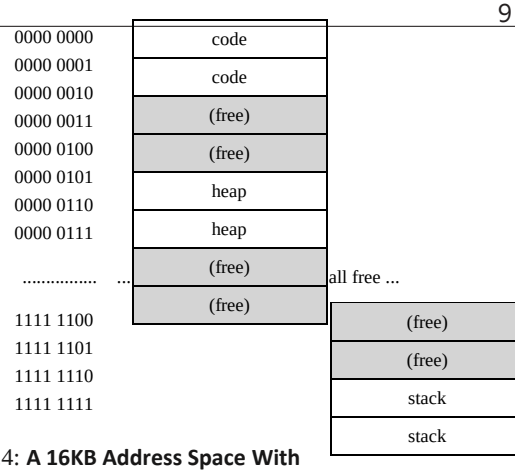
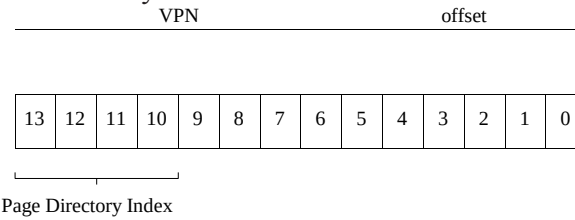


Figure 20.4: **A 16KB Address Space With 64-byte Pages**

Böylece sayfa tablomuz 1KB (256×4 byte) boyutunda oluyor. 64 baytlık sayfalarımız olduğuna göre, 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 page directory (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:

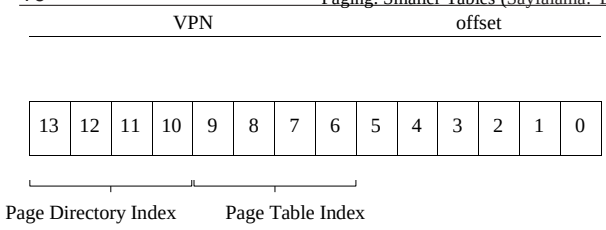


Page-directory index (Sayfa dizini dizisini) (kısaca PDIndex) çıkardıktan sonra

VPN'de basit bir hesaplama ile 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.

Page-directory index (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 bu **Sayfa dizini** girişi tarafından işaret edilen

sayfasından sayfa tablosu girişini (PTE) getirmemiz gerekiyor. Bu PTE'yi bulmak için, VPN'nin kalan bitlerini kullanarak sayfa tablosunun bölümüne i yazılır.



Bu **page-table index (sayfa tablosu dizini)** (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 gerektiğini unutmayın .

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 **page directory (sayfa dizini)** ile başlayalım (Şekil 20.5'in sol tarafı).

Şekilde, her **page directory (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 Şekil 20.5'e (orta kısım) bakınız.

Page Directory		**Page of PT (@PFN:100)**			**Page of PT (@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-

Figure 20.5: **A Page Directory, And Pieces Of Page Table**

TIP: BE WARY OF COMPLEXITY

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, olabildiğince 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 ünlü bir şekilde yazdığı gibi: "Mükemmelliğe, artık 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 işaretlenir.

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

Örnekte, VPN 254 ve 255 (yığın (the stack)) geçerli eşlemelere sahiptir. Umarız, bu örnekte görebildiğimiz şey, **Multi-level Page Tables (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 **page directory (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.

.

Page directory (sayfa dizini) indekslemek için VPN'nin ilk 4 bitini kullanacağımızı hatırlayın. Böylece, 1111, yukarıdaki **page directory (sayfa dizinin)** 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 55. sayfada eşlendiğini söyler. PFN=55 (veya hex 0x37) ile offset=000000 birleştirilerek , böylece istenen fiziksel adresimizi oluşturabilir ve talebi bellek sistemine gönderebiliriz: PhysAddr = (PTE.PFN << SHIFT) + offset = 00 1101 1100 0000 = 0x0DC0.

More Than Two Levels (İkiden Fazla Düzey)

Şimdiye kadarki örneğimizde, **Multi-level Page Tables (çok düzeyli)**sayfa tablolarının yalnızca iki düzeyi olduğunu varsaydık: bir **page directory (sayfa dizini)**ve ardından sayfa tablosunun parçaları. Bazı durumlarda, daha derin bir ağaç mümkündür (ve aslında gereklidir).

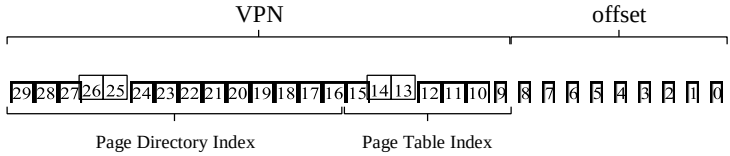
Basit bir örnek alıp daha derin bir **Multi-level Page Tables (çok düzeyli)** tablonun neden yararlı olabileceğini göstermek için kullanalım. Bu örnekte, 30 bit sanal adres alanımız ve küçük (512 bayt) bir sayfamız

olduğunu varsayalım. Böylece sanal adresimiz 21 bitlik bir sanal sayfa numarası bileşenine ve 9 bitlik bir kaymaya sahiptir.

Multi-level Page Tables (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?

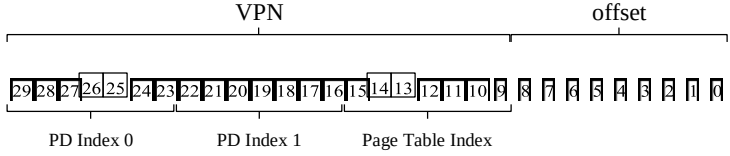
Multi-level Page Tables (çok düzeyli)bir tabloda kaç düzeyin gerekli olduğunu belirlemek için

sayfa tablosunun tüm parçalarını bir sayfaya sığdırmak için, bir sayfaya kaç tane 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 şemadan da fark edebileceğiniz bir şey (büyük) **page directory (sayfa dizini)**kaç bit kaldığıdır: 14. **page directory (sayfa dizini)** 2^{14} giriş varsa, bir sayfa değil 128 sayfaya yayılır ve bu nedenle her parçayı yapma hedefimiz **Multi-level Page Tables (çok düzeyli)** sayfa tablosunun bir sayfaya sığması kaybolur.

Bu sorunu çözmek için, **page directory (sayfa dizini)**kendisini birden çok sayfaya bölerek ve ardından **page directory (sayfa dizini)**sayfalarını işaret etmek için bunun üstüne başka bır **page directory (sayfa dizini)**ekleyerek ağacın daha ileri bir seviyesini oluşturuyoruz. Böylece sanal adresimizi aşağıdaki gibi bölebiliriz:



Şimdi, upper-level page directory (üst düzey sayfa dizini dizini oluştururken), sanal adresin en üstteki bitlerini kullanıyoruz (şemada PD Index 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 Index 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()
```

Figure 20.6: **Multi-level Page Table Control Flow: (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.

The Translation Process: Remember the TLB (Çevirici Süreci: TLB'yi unutmayın)

two-level page table (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 **Multi-level Page Tables (çok düzeyli)** erişimi gerçekleşmeden önce, donanım önce TLB'yi kontrol eder; bir isabet üzerine, physical address fiziksel adres, daha önce olduğu gibi, sayfa tablosuna hiç erişilmeden doğrudan oluşturulur. Donanımın tam **Multi-level Page Tables (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 Inverted Page Tables (Ters Sayfa Tabloları)

Bu **,inverted page tables (Tersine çevrilmiş sayfa tabloları)**, sayfa tabloları dünyasında daha da fazla yer tasarrufu sağlar. Burada, birçok sayfaya sahip olmak yerine tablolar (sistemin her işlemi için bir tane), sistemin her fiziksel sayfası için bir girişi olan tek sayfalık bir tablo 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 yapılarıdır. Veri yapılarıyla onları daha küçük veya daha büyük, daha yavaş veya daha hızlı hale getirerek pek çok çılgınca şey yapabilirsiniz. **Multi-level Page Tables (çok düzeyli)**v e ters çevrilmiş sayfa tabloları, yapılabilecek pek çok şeyin yalnızca iki örneğidir.

20.5 Swapping the Page Tables to Disk (Sayfa Tablolarını Diske Değiştirme)

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

20.6 Summary

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 sistemde TLB kayıplarını hızlandıran daha büyük bir tablo doğru seçim olabilir. Yazılımla yönetilen TLB'lerle, 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.

Think of these questions as you fall asleep, and dream the big dreams that only operating-system developers can dream.

References

[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, **Multi-level Page Tables (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 README'ye bakın.

Sorular

Soru 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?Cevap1. Donanım ilk sayfa tablosuna dizine eklendiğinde, sonraki sayfa tablosunun adresini alır. 32 bit sanal adres alanındaki olası her sayfa tablosu için bir kayıt gerektirmek pratik olmayacaktır. Cevap 1 dir.

Soru 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?

CEVAP 2. `./paging-multilevel-translate.py -s 0`

```
Virtual Address 611c:
  --> pde index:0x18 [decimal 24] pde contents:0xa1 (valid 1, pfn 0x21 [decimal 33])
    --> pte index:0x08 [decimal 8] pte contents:0xb5 (valid 1, pfn 0x35 [decimal 53])
      --> Translates to Physical Address 0x6bc --> Value: 0x08
Virtual Address 3da8:
  --> pde index:0x0f [decimal 15] pde contents:0xd6 (valid 1, pfn 0x56 [decimal 86])
    --> pte index:0x0c [decimal 13] pte contents:0x00 (valid 0)
      --> Fault
Virtual Address 17f5:
  --> pde index:0x05 [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 7f6c:
  --> pde index:0x1f [decimal 31] pde contents:0xff (valid 1, pfn 0x7f [decimal 127])
    --> pte index:0x1b [decimal 27] pte contents:0x7f (valid 0)
      --> Fault
Virtual Address 0bad:
  --> pde index:0x02 [decimal 2] pde contents:0xe0 (valid 1, pfn 0x60 [decimal 96])
    --> pte index:0x1d [decimal 29] pte contents:0x7f (valid 0)
      --> Fault
Virtual Address 6d60:
  --> pde index:0x1b [decimal 27] pde contents:0xc2 (valid 1, pfn 0x42 [decimal 66])
    --> pte index:0x0b [decimal 11] pte contents:0x7f (valid 0)
      --> Fault
Virtual Address 2a5b:
  --> pde index:0x0a [decimal 10] pde contents:0xd5 (valid 1, pfn 0x55 [decimal 85])
    --> pte index:0x12 [decimal 18] pte contents:0x7f (valid 0)
      --> Fault
Virtual Address 4c5e:
  --> pde index:0x13 [decimal 19] pde contents:0xf8 (valid 1, pfn 0x78 [decimal 120])
    --> pte index:0x02 [decimal 2] pte contents:0x7f (valid 0)
      --> Fault
Virtual Address 2592:
  --> pde index:0x09 [decimal 9] pde contents:0x9e (valid 1, pfn 0x1e [decimal 30])
    --> pte index:0x0c [decimal 12] pte contents:0xbd (valid 1, pfn 0x3d [decimal 61])
      --> Translates to Physical Address 0x7b2 --> Value: 0x1b
Virtual Address 3e99:
  --> pde index:0x0f [decimal 15] pde contents:0xd6 (valid 1, pfn 0x56 [decimal 86])
    --> pte index:0x14 [decimal 20] pte contents:0xca (valid 1, pfn 0x4a [decimal 74])
      --> Translates to Physical Address 0x959 --> Value: 0x1e
```

- `./paging-multilevel-translate.py -s 1 -n 5`

```
Virtual Address 6c74:
  --> pde index:0x1b [decimal 27] pde contents:0xa0 (valid 1, pfn 0x20 [decimal 32])
    --> pte index:0x02 [decimal 2] pte contents:0x7f (valid 0)      /* incorrect */
      --> Fault
Virtual Address 6b22:
  --> pde index:0x1a [decimal 26] pde contents:0xd2 (valid 1, pfn 0x52 [decimal 82])
    --> pte index:0x19 [decimal 25] pte contents:0xc7 (valid 1, pfn 0x47 [decimal 71])
      --> Translates to Physical Address 0x8e2 --> Value: 0x1a
Virtual Address 03df:
  --> pde index:0x00 [decimal 0] pde contents:0xda (valid 1, pfn 0x5a [decimal 90])
    --> pte index:0x1e [decimal 30] pte contents:0x85 (valid 1, pfn 0x05 [decimal 5])
      --> Translates to Physical Address 0x0bf --> Value: 0x0f
Virtual Address 69dc:
  --> pde index:0x1a [decimal 26] pde contents:0xd2 (valid 1, pfn 0x52 [decimal 82])
    --> pte index:0x0e [decimal 14] pte contents:0x7f (valid 0)
      --> Fault
Virtual Address 317a:
  --> pde index:0x0c [decimal 12] pde contents:0x98 (valid 1, pfn 0x18 [decimal 24])
    --> pte index:0x0b [decimal 11] pte contents:0xb5 (valid 1, pfn 0x35 [decimal 53])
      --> Translates to Physical Address 0x6ba --> Value: 0x1e
```

- `./paging-multilevel-translate.py -s 2 -n 5`

```
Virtual Address 7570:
  --> pde index:0x1d [decimal 29] pde contents:0xb3 (valid 1, pfn 0x33 [decimal 51])
    --> pte index:0x0b [decimal 11] pte contents:0x7f (valid 0)
      --> Fault
Virtual Address 7268:
  --> pde index:0x1c [decimal 28] pde contents:0xde (valid 1, pfn 0x5e [decimal 94])
    --> pte index:0x13 [decimal 19] pte contents:0xe5 (valid 1, pfn 0x65 [decimal 101])
      --> Translates to Physical Address 0xca8 --> Value: 0x16
Virtual Address 1f9f:
  --> pde index:0x07 [decimal 7] pde contents:0xaf (valid 1, pfn 0x2f  [decimal 47])
    --> pte index:0x1c [decimal 28] pte contents:0x7f (valid 0)
      --> Fault
Virtual Address 0325:
  --> pde index:0x00 [decimal 0] pde contents:0x82 (valid 1, pfn 0x02  [decimal 2])
    --> pte index:0x19 [decimal 25] pte contents:0xdd (valid 1, pfn 0x5d [decimal 93])
      --> Translates to Physical Address 0xba5 --> Value: 0x0b
Virtual Address 64c4:
  --> pde index:0x19 [decimal 25] pde contents:0xb8 (valid 1, pfn 0x38  [decimal 56])
    --> pte index:0x06 [decimal 6] pte contents:0x7f (valid 0)
      --> Fault
```

CEVAP 3 ; Önbelleğin nasıl çalıştığını anladığınıza göre, sayfa tablosuna yapılan bellek başvurularının 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?
Cevap3.Bu alıştırmalardaki adresler rastgele olduğu için ıskalama oranı oldukça yüksek olacaktır. Eğer isabet oranı yüksek olacaksa TLB kullanımının amacı olmazdı.