19

Paging: Faster Translations (TLBs)

Disk belleğini sanal belleği desteklemek için çekirdek mekanizma olarak kullanmak, yüksek performans giderlerine yol açabilir. Adres alanını küçük, sabit boyutlu birimlere (yani sayfalara) bölen sayfalama, büyük miktarda eşleme bilgisi gerektirir. Bu eşleme bilgileri genellikle fiziksel bellekte depolandığından, sayfalama mantıksal olarak program tarafından oluşturulan her sanal adres için fazladan bir bellek araması gerektirir. Her talimat getirmeden veya açık yüklemeden veya depolamadan önce çeviri bilgileri için belleğe gitmek engelleyici derecede yavaştır. Ve böylece sorunumuz:

THE CRUX:
HOW TO SPEED UP ADDRESS TRANSLATION

Adres çevirisini nasıl hızlandırabiliriz ve genellikle sayfalamanın gerektirdiği ek bellek referansından nasıl kaçınabiliriz? Hangi donanım desteği gereklidir? Hangi işletim sistemi katılımı gereklidir?

İşleri hızlandırmak istediğimizde, işletim sisteminin genellikle biraz yardıma ihtiyacı vardır. Ve yardım genellikle işletim sisteminin eski dostundan gelir: donanım. Adres çevirisini hızlandırmak için, (tarihsel nedenlerden dolayı [CP78]) bir çeviri-görünüm arabelleği **(translation-lookaside buffer)** ekleyeceğiz, yada **TLB** [CG68, C95]. TLB, çipin bellek yönetimi biriminin **(memory-management unit (MMU))** bir parçasıdır, ve sanaldan fiziksele adres çevirilerinin popüler bir donanım önbelleğidir; bu nedenle, adres çeviri önbelleği **(address-translation cache)** daha iyi bir ad olacaktır. Her bir sanal bellek referansında, donanım önce istenen çevirinin burada tutulup tutulmadığını görmek için TLB'yi kontrol eder; öyleyse, çeviri (hızlı bir şekilde) sayfa tablosuna (tüm çevirileri içeren) başvurmak zorunda kalmadan gerçekleştirilir. Muazzam performans etkileri nedeniyle, TLB'ler gerçek anlamda sanal belleği mümkün kılar [C95].

```
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     PTEAddr = PTBR + (VPN * sizeof(PTE))
12     PTE = AccessMemory(PTEAddr)
13     if (PTE.Valid == False)
14         RaiseException(SEGMENTATION_FAULT)
15     else if (CanAccess(PTE.ProtectBits) == False)
16         RaiseException(PROTECTION_FAULT)
17     else
18         TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
19         RetryInstruction()
```

Figure 19.1: **TLB Control Flow Algorithm**

19.1 TLB Basic Algorithm

Şekil 19.1, basit bir doğrusal sayfa tablosu **(linear page table)** ve donanım tarafından yönetilen bir TLB **(hardware-managed TLB)** (yani, sayfanın sorumluluğunun çoğunu donanım üstlenir) varsayarak, donanımın bir sanal adres çevirisini nasıl işleyebileceğinin kabaca bir taslağını gösterir. tablo erişimleri; bununla ilgili daha fazla bilgiyi aşağıda açıklayacağız).

Donanımın izlediği algoritma şu şekilde çalışır: önce, sanal adresten sanal sayfa numarasını (VPN) çıkarın (Şekil 19.1'deki Satır 1) ve TLB'nin bu VPN için çeviriyi elinde tutup tutmadığını kontrol edin (Satır 2). Varsa, bir TLB isabetimiz var, bu da TLB'nin çeviriyi elinde tuttuğu anlamına gelir. Başarı! Artık sayfa çerçeve numarasını (PFN) ilgili TLB girişinden çıkarabilir, bunu orijinal sanal adresten ofset üzerinde birleştirebilir ve istenen fiziksel adresi (PA) oluşturabilir ve korumayı varsayarak belleğe erişebilir (Satır 5-7) kontroller başarısız olmaz (Satır 4).

CPU çeviriyi TLB'de bulamazsa (bir TLB eksikliği), yapacak daha çok işimiz var. Bu örnekte, donanım çeviriyi bulmak için sayfa tablosuna erişir (Satır 11–12) ve işlem tarafından oluşturulan sanal bellek referansının geçerli ve erişilebilir olduğunu varsayarak (Satır 13, 15), TLB'yi şu şekilde günceller: çeviri (Satır 18). Bu eylemler dizisi, öncelikle sayfa tablosuna (Satır 12) erişmek için gereken ekstra bellek referansı nedeniyle maliyetlidir. Son olarak, TLB güncellendiğinde, donanım komutu yeniden dener; bu kez çeviri TLB'de bulunur ve bellek referansı hızlı bir şekilde işlenir.

TLB, tüm önbellekler gibi, genel durumda çevirilerin önbellekte bulunduğu (yani, isabetler olduğu) öncülü üzerine inşa edilmiştir. Eğer öyleyse, TLB işlemci çekirdeğinin yakınında bulunduğundan ve oldukça hızlı olacak şekilde tasarlandığından, çok az ek yük eklenir. Bir hata oluştuğunda, yüksek sayfalama maliyeti ortaya çıkar; çeviriyi bulmak için sayfa tablosuna erişilmelidir ve fazladan bir bellek referansı (veya daha karmaşık sayfa tablolarıyla daha fazlası) oluşur. Bu sık sık meydana gelirse, program büyük olasılıkla fark edilir şekilde daha yavaş çalışacaktır; çoğu CPU talimatına göre bellek erişimleri oldukça maliyetlidir ve TLB kayıplar daha fazla bellek erişimine yol açar. Bu nedenle, elimizden geldiğince TLB ıskalamalarından kaçınmayı umuyoruz.

19.2 Example: Accessing An Array

Bir TLB'nin işleyişini netleştirmek için basit bir sanal adres izlemeyi inceleyelim ve bir TLB'nin performansını nasıl artırabileceğini görelim. Bu örnekte, bellekte sanal adres 100'den başlayan 10 adet 4 baytlık bir tamsayı dizimiz olduğunu varsayalım. Ayrıca, 16 baytlık sayfaları olan küçük bir 8 bitlik sanal adres alanımız olduğunu varsayalım; bu nedenle, bir sanal adres 4-bit VPN (16 sanal sayfa vardır) ve 4-bit ofset (bu sayfaların her birinde 16 bayt vardır) olarak bölünür.

Şekil 19.2 (sayfa 4), sistemin 16 adet 16 baytlık sayfasında düzenlenen diziyi göstermektedir. Gördüğünüz gibi dizinin ilk girişi (a[0]) (VPN=06, offset=04) ile başlıyor; o sayfaya yalnızca üç adet 4 baytlık tamsayı sığar. Dizi, sonraki dört girişin (a[3] ... a[6]) bulunduğu sonraki sayfaya (VPN=07) devam eder. Son olarak, 10 girişli dizinin (a[7] ... a[9]) son üç girişi, adres alanının (VPN=08) bir sonraki sayfasında bulunur.

Şimdi her bir dizi öğesine erişen basit bir döngü düşünelim, C'de şöyle görünecek bir şey:

```
int sum = 0;
for (i = 0; i < 10; i++) {
    sum += a[i];
}
```

Basitlik adına, döngünün ürettiği tek bellek erişiminin diziye olduğunu farz edeceğiz (i ve sum değişkenlerinin yanı sıra talimatların kendilerini de göz ardı ederek). İlk dizi öğesine (a[0]) erişildiğinde, CPU sanal adres 100'e bir yük görecektir. Donanım bundan VPN'i çıkarır (VPN=06) ve bunu TLB'yi geçerli bir çeviri için kontrol etmek için kullanır. Programın diziye ilk kez eriştiği varsayılırsa, sonuç bir TLB hatası olacaktır.

Bir sonraki erişim a[1]'e olacak ve burada bazı iyi haberler var: bir TLB vuruşu! Dizinin ikinci öğesi birinci öğenin yanında paketlendiği için aynı sayfada yer alır; dizinin ilk öğesine erişirken bu sayfaya zaten eriştiğimiz için çeviri zaten yüklendi

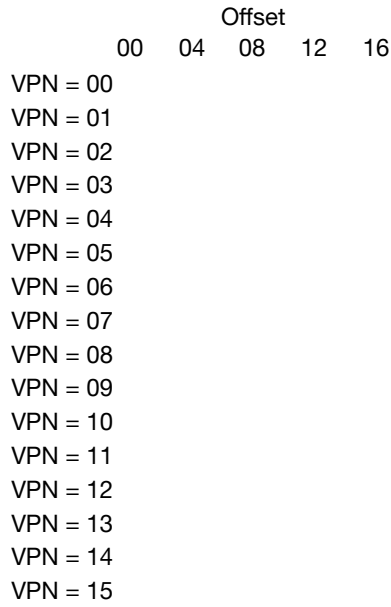


Figure 19.2: **Example: An Array In A Tiny Address Space**

TLB'ye girin. Ve dolayısıyla başarımızın nedeni. a[2]'ye erişim benzer bir başarı ile karşılaşır (başka bir isabet), çünkü o da a[0] ve a[1] ile aynı sayfada yaşar.

Ne yazık ki, program a[3]'e eriştiğinde, başka bir TLB hatasıyla karşılaşıyoruz. Ancak, bir kez daha, sonraki girişler (a[4] ... a[6]), tümü bellekte aynı sayfada bulunduğu için TLB'de bulunacaktır.

Son olarak, a[7]'ye erişim, son bir TLB'nin kaçırılmasına neden olur. Donanım, bu sanal sayfanın fiziksel bellekteki yerini bulmak için bir kez daha sayfa tablosuna başvurur ve TLB'yi buna göre günceller. Son iki erişim (a[8] ve a[9]) bu TLB güncellemesinin faydalarını alır; donanım, çevirileri için TLB'ye baktığında, iki isabet daha elde edilir.

Diziye on erişimimiz sırasındaki TLB etkinliğini özetleyelim: **miss, hit, hit, miss, hit, hit, hit, miss, hit, hit.**. Böylece, isabet sayısının toplam erişim sayısına bölünmesiyle elde edilen TLB isabet oranımız %70'tir. Bu çok yüksek olmasa da (aslında %100'e yaklaşan isabet oranlarını arzuluyoruz), sıfır değil ki bu sürpriz olabilir. Programın diziye ilk kez erişmesine rağmen, TLB, uzamsal konum nedeniyle performansı artırır. Dizinin öğeleri, sayfalar halinde sıkıca paketlenir (yani, uzayda birbirlerine yakındırlar) ve bu nedenle, bir sayfadaki bir öğeye yalnızca ilk erişim, **TLB**'nin kaybolmasına neden olur.

Ayrıca bu örnekte sayfa boyutunun oynadığı role dikkat edin. Eğer sayfa boyutu

TIP: USE CACHING WHEN POSSIBLE

Önbelleğe alma, bilgisayar sistemlerindeki en temel performans tekniklerinden biridir ve "sıradan durumu hızlı" hale getirmek için tekrar tekrar kullanılır [HP06]. Donanım önbelleklerinin arkasındaki fikir, talimat ve veri referanslarında yerellikten yararlanmaktır. Genellikle iki tür yerellik vardır: zamansal yerellik **(temporal locality)** ve mekansal yerellik **(spatial locality)**. Zamansal yerellik ile, yakın zamanda erişilen bir yönerge veya veri öğesine gelecekte yakında yeniden erişilmesi muhtemeldir. Döngü değişkenlerini veya bir döngüdeki yönergeleri düşünün; zaman içinde tekrar tekrar erişilirler. Uzamsal yerellik ile, bir program x adresindeki belleğe erişirse, muhtemelen yakında x yakınındaki belleğe erişecektir. Burada bir tür diziden akış yaptığınızı, bir öğeye ve ardından diğerine eriştiğinizi hayal edin. Tabii ki, bu özellikler programın tam doğasına bağlıdır ve bu nedenle katı ve hızlı yasalar değil, daha çok pratik kurallar gibidir.

Talimatlar, veriler veya adres çevirileri için (TLB'mizde olduğu gibi) donanım önbellekleri, belleğin kopyalarını küçük, hızlı çip üzerinde bellekte tutarak yerellikten yararlanır. Bir isteği yerine getirmek için (yavaş) bir belleğe gitmek yerine, işlemci önce önbellekte yakındaki bir kopyanın olup olmadığını kontrol edebilir; eğer öyleyse, işlemci ona hızlı bir şekilde erişebilir (yani, birkaç CPU döngüsünde) ve belleğe erişmek için gereken maliyetli zamanı (birçok nanosaniye) harcamaktan kaçınabilir.

Merak ediyor olabilirsiniz: Önbellekler (TLB gibi) bu kadar harikaysa, neden daha büyük önbellekler oluşturup tüm verilerimizi içlerinde tutmuyoruz? Ne yazık ki, fiziğinkiler gibi daha temel yasalarla karşılaştığımız yer burasıdır. Hızlı bir önbellek istiyorsanız, ışık hızı ve diğer fiziksel kısıtlamalar önemli hale geldiğinden, küçük olması gerekir. Tanımı gereği herhangi bir büyük önbellek yavaştır ve bu nedenle amacı bozar. Bu nedenle, küçük, hızlı önbelleklerle sıkışıp kaldık; Geriye kalan soru, performansı artırmak için bunları en iyi nasıl kullanacağımızdır.

basitçe iki kat daha büyük olsaydı (16 değil, 32 bayt), dizi erişimi daha da az kayıp yaşardı. Tipik sayfa boyutları daha çok 4 KB gibi olduğundan, bu tür yoğun, dizi tabanlı erişimler, her sayfa erişimde yalnızca tek bir kayıpla karşılaşarak mükemmel TLB performansı sağlar.

TLB performansıyla ilgili son bir nokta: eğer program, bu döngü tamamlandıktan hemen sonra diziye tekrar erişirse, gerekli çevirileri önbelleğe alacak kadar büyük bir TLB'ye sahip olduğumuzu varsayarsak, muhtemelen daha da iyi bir sonuç görürüz: **hit,hit,hit,hit,hit,hit,hit,hit,hit,hit**. Bu durumda TLB isabet oranı, zamansal konum nedeniyle, yani bellek öğelerinin zaman içinde hızlı bir şekilde yeniden referanslandırılması nedeniyle yüksek olacaktır. Herhangi bir önbellek gibi, TLB'ler de başarı için program özellikleri olan hem uzamsal hem de zamansal konuma güvenir. İlgili program bu tür yerellik **(temporal locality)** sergiliyorsa (ve birçok program gösteriyorsa), TLB isabet oranı muhtemelen yüksek olacaktır.

```
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     RaiseException(TLB_MISS)
```

Figure 19.3: **TLB Control Flow Algorithm (OS Handled)**

19.3 Who Handles The TLB Miss?

OCevaplamamız gereken bir soru: Bir TLB özlediğini kim halleder? İki cevap mümkündür: donanım veya yazılım (OS). Eski günlerde, donanımın karmaşık komut setleri (bazen **CISC** olarak adlandırılır) vardı., karmaşık komut setli bilgisayarlar için) ve donanımı oluşturan kişiler, bu sinsi işletim sistemi çalışanlarına pek güvenmezdi. Böylece donanım, TLB eksikliğini tamamen giderebilir. Bunu yapmak için, donanımın sayfa tablolarının bellekte tam olarak nerede bulunduğunu bilmesi gerekir (Şekil 19.1'de Satır 11'de kullanılan bir sayfa tablosu temel kaydı **(page- table base register)** yoluyla), tam biçimlerinin yanı sıra; ıskalama durumunda, donanım sayfa tablosunu "yürütür", doğru sayfa tablosu girişini bulur ve istenen çeviriyi çıkarır, TLB'yi çeviriyle günceller ve talimatı yeniden dener. Donanım tarafından yönetilen TLB'lere **(hardware-managed)** sahip "eski" bir mimari örneği, sabit bir çok düzeyli sayfa tablosu kullanan Intel x86 mimarisidir (ayrıntılar için bir sonraki bölüme bakın); geçerli sayfa tablosu CR3 kaydı [I09] tarafından işaret edilir.

Daha modern mimariler (örneğin, MIPS R10k [H93] veya Sun'ın SPARC v9 [WG00], her ikisi de **RISC** veya azaltılmış komut setli bilgisayarlar) yazılım tarafından yönetilen TLB olarak bilinen şeye sahiptir. Bir TLB hatasında donanım basitçe bir istisna oluşturur (Şekil 19.3'teki 11. satır), mevcut talimat akışını duraklatır, ayrıcalık seviyesini çekirdek moduna yükseltir ve bir tuzak işleyiciye atlar. Tahmin edebileceğiniz gibi, bu tuzak işleyici, TLB kayıplarını işlemek amacıyla yazılmış işletim sistemi içindeki koddur. Çalıştırıldığında, kod sayfa tablosundaki çeviriyi arayacak, TLB'yi güncellemek için özel "ayrıcalıklı" yönergeleri kullanacak ve tuzaktan geri dönecektir; bu noktada, donanım talimatı yeniden dener (bir TLB isabetiyle sonuçlanır). Birkaç önemli ayrıntıyı tartışalım. Birincisi, tuzaktan dönüş talimatının, daha önce bir sistem çağrısına hizmet verirken gördüğümüz tuzaktan dönüş komutundan biraz farklı olması gerekir. İkinci durumda, tıpkı bir prosedür çağrısından dönüşün prosedür çağrısından hemen sonra talimata geri dönmesi gibi, tuzaktan dönüş OS'ye tuzaktan sonra talimatta yürütmeye devam etmelidir. Önceki durumda, bir TLB yanlış işleme tuzağından dönerken, donanım tuzağa neden olan talimatta yürütmeye devam etmelidir; bu yeniden deneme böylece girişi sağlar

ASIDE: RISC VS. CISC

1980'lerde bilgisayar mimarisi camiasında büyük bir savaş yaşandı. Bir tarafta, Karmaşık Komut Seti Hesaplamanın**(Complex Instruction Set Computing)** kısaltması olan **CISC** kampı vardı; diğer tarafta, Azaltılmış Komut Seti Hesaplama**(Reduced Instruction Set Computing)** [PS81] için **RISC** vardı. RISC tarafı, Berkeley'den David Patterson ve Stanford'dan John Hennessy (aynı zamanda bazı ünlü kitapların ortak yazarları [HP06]) tarafından yönetildi, ancak daha sonra John Cocke, RISC üzerine ilk çalışması nedeniyle bir Turing ödülü ile tanındı [CM00].

CISC komut setlerinde çok sayıda talimat bulunur ve her talimat nispeten güçlüdür. Örneğin, iki işaretçi ve bir uzunluk alan ve baytları kaynaktan hedefe kopyalayan bir dize kopyası görebilirsiniz. CISC'nin arkasındaki fikir, montaj dilinin kendisinin kullanımını kolaylaştırmak ve kodu daha kompakt hale getirmek için talimatların üst düzey ilkel olması gerektiğiydi.

RISC komut setleri tam tersidir. RISC'nin arkasındaki önemli bir gözlem, komut setlerinin gerçekten derleyici hedefleri olduğu ve derleyicilerin gerçekten istediği tek şeyin, yüksek performanslı kod oluşturmak için kullanabilecekleri birkaç basit ilkel olmasıdır. Bu nedenle, RISC savunucuları, donanımdan mümkün olduğu kadar çok şeyi (özellikle mikro kodu) söküp atalım ve geriye kalanları basit, tekdüze ve hızlı hale getirelim.

İlk günlerde, RISC çipleri fark edilir derecede daha hızlı oldukları için büyük bir etki yarattı [BC91]; birçok makale yazıldı; birkaç şirket kuruldu (örneğin, MIPS ve Sun). Bununla birlikte, zaman geçtikçe, Intel gibi CISC üreticileri, örneğin karmaşık talimatları daha sonra RISC benzeri bir şekilde işlenebilen mikro talimatlara dönüştüren erken boru hattı aşamaları ekleyerek, birçok RISC tekniğini işlemcilerinin çekirdeğine dahil ettiler. . Bu yeniliklere ek olarak her çipte artan sayıda transistör, CISC'nin rekabetçi kalmasını sağladı. Sonuç olarak, tartışma sona erdi ve bugün her iki işlemci türü de hızlı çalışacak şekilde yapılabilir.

.

yapı yeniden çalışır, bu sefer bir TLB isabetiyle sonuçlanır. Bu nedenle, bir tuzağın veya istisnanın nasıl oluşturulduğuna bağlı olarak, işletim sistemine bindirme sırasında donanımın zamanı geldiğinde düzgün bir şekilde devam etmesi için farklı bir PC'yi kaydetmesi gerekir.

İkincisi, TLB hata işleme kodunu çalıştırırken, işletim sisteminin sonsuz sayıda TLB hatalarının oluşmasına neden olmamak için ekstra dikkatli olması gerekir. Birçok çözüm mevcuttur; örneğin, TLB hata işleyicilerini fiziksel bellekte tutabilirsiniz (burada eşlenmemişler ve adres çevirisine tabi değiller) veya TLB'deki bazı girişleri kalıcı olarak geçerli çeviriler için ayırabilir ve bu kalıcı çevirilerden bazılarını kullanabilirsiniz. işleyici kodunun kendisi için yuvalar; bu kablolu çeviriler her zaman TLB'de görülür.

Yazılımla yönetilen yaklaşımın birincil avantajı esnekliktir: İşletim sistemi, sayfayı uygulamak istediği herhangi bir veri yapısını kullanabilir.

ASIDE: TLB VALID BIT /= PAGE TABLE VALID BIT

Bir TLB'de bulunan geçerli bitleri bir sayfa tablosunda bulunanlarla karıştırmak yaygın bir hatadır. Bir sayfa tablosunda, bir sayfa tablosu girişi (PTE) geçersiz olarak işaretlendiğinde, bu, sayfanın işlem tarafından tahsis edilmediği ve düzgün çalışan bir program tarafından erişilmemesi gerektiği anlamına gelir. Geçersiz bir sayfaya erişildiğinde olağan yanıt, işlemi sonlandırarak yanıt verecek olan işletim sistemine tuzak kurmaktır.

Bir TLB geçerli biti, aksine, basitçe bir TLB girişinin içinde geçerli bir çeviriye sahip olup olmadığını ifade eder. Örneğin, bir sistem önyüklendiğinde, her TLB girişi için ortak bir başlangıç durumu geçersiz olarak ayarlanmalıdır, çünkü burada henüz hiçbir adres çevirisi önbelleğe alınmamıştır. Sanal bellek etkinleştirildikten ve programlar çalışmaya ve sanal adres alanlarına erişmeye başladığında, TLB yavaş yavaş doldurulur ve bu nedenle geçerli girişler kısa süre sonra TLB'yi doldurur.

TLB geçerli biti, aşağıda daha ayrıntılı olarak tartışacağımız gibi, bir içerik anahtarı gerçekleştirirken de oldukça kullanışlıdır. Sistem, tüm TLB girişlerini geçersiz olarak ayarlayarak, çalıştırılmak üzere olan işlemin yanlışlıkla önceki bir işlemden sanaldan fiziksele çeviriyi kullanmamasını sağlayabilir.

tablo, donanım değişikliği gerektirmeden. Diğer bir avantaj, TLB kontrol akışında görüldüğü gibi basitliktir (Şekil 19.1'deki 11–19 satırlarının aksine Şekil 19.3'teki 11. satır). Donanım ıskalama durumunda fazla bir şey yapmaz: sadece bir istisna oluşturun ve OS TLB'nin ıskalama işleyicisinin gerisini halletmesine izin verin.

19.4 TLB Contents: What's In There?

Donanım TLB'sinin içeriğine daha detaylı bakalım. Tipik bir TLB'nin 32, 64 veya 128 girişi olabilir ve tamamen çağrışımsal denilen şey olabilir. Temel olarak bu, herhangi bir çevirinin TLB'de herhangi bir yerde olabileceği ve donanımın**(fully associa- tive)** istenen çeviriyi bulmak için tüm TLB'yi paralel olarak arayacağı anlamına gelir. Bir TLB girişi şöyle görünebilir:

VPN PFN other bits

Bir çeviri bu konumlardan herhangi birinde sona erebileceğinden (donanım terimleriyle, TLB tamamen ilişkilendirilebilir **(fully-associative)** bir önbellek olarak bilinir) her girişte hem VPN hem de PFN'nin mevcut olduğuna dikkat edin. Donanım, bir eşleşme olup olmadığını görmek için girişleri paralel olarak arar.

Daha ilginç olan "diğer parçalar". Örneğin, TLB'nin genellikle, girişin geçerli bir çevirisi olup olmadığını söyleyen geçerli bir biti vardır. Ayrıca bir sayfaya nasıl erişilebileceğini belirleyen (sayfa tablosundaki gibi) koruma bitleri de yaygındır. Örneğin, kod sayfaları okundu ve yürütüldü olarak işaretlenebilirken yığın sayfaları okundu ve yazıldı olarak işaretlenebilir.

19.5 TLB Issue: Context Switches

TLB'lerde, işlemler (ve dolayısıyla adres alanları) arasında geçiş yaparken bazı yeni sorunlar ortaya çıkar. Spesifik olarak, TLB, yalnızca o anda çalışan işlem için geçerli olan sanaldan fiziksele çeviriler içerir; bu çeviriler diğer süreçler için anlamlı değildir. Sonuç olarak, bir işlemden diğerine geçerken, donanım veya işletim sistemi (veya her ikisi) çalıştırılmak üzere olan işlemin önceden çalıştırılan bazı işlemlerden çevirileri yanlışlıkla kullanmadığından emin olun.

Bu durumu daha iyi anlamak için bir örneğe bakalım. Bir işlem (P1) çalışırken, TLB'nin kendisi için geçerli olan, yani P1'in sayfa tablosundan gelen çevirileri önbelleğe aldığını varsayar. Bu örnek için, P1'in 10. sanal sayfasının fiziksel çerçeve 100 ile eşlendiğini varsayalım. Bu örnekte, başka bir işlemin (P2) var olduğunu ve işletim sisteminin yakında bir bağlam anahtarı gerçekleştirmeye ve onu çalıştırmaya karar verebileceğini varsayalım. Burada P2'nin 10. sanal sayfasının fiziksel çerçeve 170'e eşlendiğini varsayalım.

her iki süreç de TLB'deydi, TLB'nin içeriği şöyle olurdu:

VPN	PFN	valid	prot
10	100	1	rwx
—	—	0	—
10	170	1	rwx
—	—	0	—

Yukarıdaki TLB'de açıkça bir sorunumuz var: VPN 10, PFN 100 (P1) veya PFN 170 (P2) olarak çevrilir, ancak donanım hangi girişin hangi işlem için olduğunu ayırt edemez. Bu nedenle, TLB'nin çoklu süreçlerde sanallaştırmayı doğru ve verimli bir şekilde desteklemesi için biraz daha çalışmamız gerekiyor. Ve böylece, bir dönüm noktası:

THE CRUX:
HOW TO MANAGE TLB CONTENTS ON A CONTEXT SWITCH

İşlemler arasında bağlam geçişinde, son işlem için TLB'deki çeviriler, çalıştırılmak üzere olan işlem için anlamlı değildir. Bu sorunu çözmek için donanım veya işletim sistemi ne yapmalıdır?

Bu sorunun bir dizi olası çözümü vardır. Yaklaşımlardan biri, TLB'yi içerik anahtarlarında basitçe yıkamak ve böylece bir sonraki işlemi çalıştırmadan önce boşaltmak. Yazılım tabanlı bir sistemde bu, açık (ve ayrıcalıklı) bir donanım talimatı ile gerçekleştirilebilir; donanım tarafından yönetilen bir TLB ile, sayfa tablosu temel kaydı değiştirildiğinde temizleme etkinleştirilebilir (işletim sisteminin bir bağlam anahtarında PTBR'yi her halükarda değiştirmesi gerektiğini unutmayın). Her iki durumda da, temizleme işlemi basitçe tüm geçerli bitleri 0'a ayarlar ve temel olarak TLB'nin içeriğini temizler.

Her bağlam anahtarında TLB'yi temizleyerek, artık çalışan bir çözüme sahibiz, çünkü bir süreç asla yanlışlıkla yanlış aktarımla karşılaşmaz.

TLB'deki ilişkiler. Ancak bunun bir bedeli vardır: Bir işlem her çalıştırıldığında, veri ve kod sayfalarına dokunurken TLB'nin gözden kaçmasına neden olmalıdır. İşletim sistemi, işlemler arasında sık sık geçiş yapıyorsa, bu maliyet yüksek olabilir.

Bu ek yükü azaltmak için bazı sistemler, TLB'nin bağlam anahtarları arasında paylaşılmasını sağlamak için donanım desteği ekler. Özellikle, bazı donanım sistemleri, TLB'de bir adres alanı tanımlayıcısı **(space identifier)**(ASID) alanı sağlar. ASID'yi bir süreç tanımlayıcısı **(process identifier)**(PID) olarak düşünebilirsiniz, ancak genellikle daha az bit içerir (örneğin, ASID için 8 bit, PID için 32 bit).

Yukarıdan örnek TLB'mizi alıp ASID'leri eklersek, süreçlerin TLB'yi kolayca paylaşabileceği açıktır: aksi halde aynı olan çevirileri ayırt etmek için yalnızca ASID alanı gereklidir. ASID alanı eklenmiş bir TLB'nin tasviri aşağıdadır:

VPN	PFN	valid	prot	ASID
10	100	1	rwx	1
—	—	0	—	—
10	170	1	rwx	2
—	—	0	—	—

Böylece, adres alanı tanımlayıcıları ile TLB, farklı işlemlerden gelen çevirileri aynı anda herhangi bir karışıklık olmadan tutabilir. Elbette donanımın çevirileri gerçekleştirmek için o anda hangi işlemin çalıştığını bilmesi gerekir ve bu nedenle işletim sisteminin bir bağlam anahtarında geçerli işlemin ASID'sine bazı ayrıcalıklı kayıtlar ayarlaması gerekir.

Bir yana, TLB'nin iki girişinin oldukça benzer olduğu başka bir durum da düşünmüş olabilirsiniz. Bu örnekte, aynı fiziksel sayfayı işaret eden iki farklı VPN'li iki farklı işlem için iki giriş vardır:

VPN	PFN	valid	prot	ASID
10	101	1	r-x	1
—	—	0	—	—
50	101	1	r-x	2
—	—	0	—	—

Bu durum, örneğin iki işlem bir sayfayı paylaştığında (örneğin bir kod sayfası) ortaya çıkabilir. Yukarıdaki örnekte, İşlem 1 fiziksel sayfa 101'i İşlem 2 ile paylaşmaktadır; P1 bu sayfayı adres alanının 10. sayfasına, P2 ise adres alanının 50. sayfasına eşler. Kod sayfalarının (ikili dosyalarda veya paylaşılan kitaplıklarda) paylaşılması, kullanımdaki fiziksel sayfaların sayısını azalttığı ve böylece bellek ek yüklerini azalttığı için yararlıdır.

19.6 Issue: Replacement Policy

Herhangi bir önbellekte ve dolayısıyla TLB'de olduğu gibi, dikkate almamız gereken bir sorun daha önbellek değiştirmedir**(cache replacement)**. Spesifik olarak, TLB'ye yeni bir giriş kurarken, eskisini değiştirmeliyiz ve dolayısıyla soru şu:

THE CRUX: HOW TO DESIGN TLB REPLACEMENT POLICY

Yeni bir TLB girişi eklediğimizde hangi TLB girişi değiştirilmelidir? Amaç, elbette, ıskalama oranını en aza indirmek**(miss rate)** (veya isabet oranını artırmak) ve böylece performansı iyileştirmektir.

değiştirilecek biri?

Sayfaları diske aktarma sorununu ele alırken bu tür politikaları biraz ayrıntılı olarak inceleyeceğiz; burada sadece birkaç tipik politikayı vurgulayacağız. Yaygın bir yaklaşım, en son kullanılan**(least-recently-used)** veya LRU girişini çıkarmaktır. LRU, yakın zamanda kullanılmamış bir girişin tahliye için iyi bir aday olabileceğini varsayarak, bellek referans akışındaki yerin avantajlarından yararlanmaya çalışır. Başka bir tipik yaklaşım, rastgele bir TLB eşlemesini çıkaran rastgele bir politika kullanmaktır. Böyle bir politika, basitliği ve köşe durum davranışlarından kaçınma yeteneği nedeniyle yararlıdır; örneğin, LRU gibi "makul" bir politika, bir program n boyutunda bir TLB ile n + 1 sayfa üzerinde döngü yaptığında oldukça mantıksız davranır; bu durumda, LRU her erişimi kaçırırken, rastgele çok daha iyi sonuç verir.

19.7 A Real TLB Entry

değiştirilecek biri?

Son olarak, kısaca gerçek bir TLB'ye bakalım. Bu örnek, yazılımla yönetilen TLB'leri kullanan modern bir sistem olan MIPS R4000'den [H93] alınmıştır; biraz basitleştirilmiş bir MIPS TLB girişi Şekil 19.4'te görülebilir.

MIPS R4000, 4KB sayfalı 32 bit adres alanını destekler. Bu nedenle, tipik sanal adresimizde 20 bit VPN ve 12 bit ofset beklerdik. Ancak TLB'de görebileceğiniz gibi VPN için sadece 19 bit var; Görünen o ki, kullanıcı adresleri adres alanının yalnızca yarısından gelecek (geri kalanı çekirdek için ayrılmış) ve bu nedenle yalnızca 19 bit VPN gerekiyor. VPN, 24 bit fiziksel çerçeve numarasına (PFN) kadar çevirir ve dolayısıyla 64 GB'a kadar (fiziksel) ana

24

bellek (2 4KB sayfa).

MIPS TLB'de birkaç ilginç parça daha var. bir küresel görüyoruz

bit (G), süreçler arasında küresel olarak paylaşılan sayfalar için kullanılır. Böylece, global bit ayarlanmışsa, ASID göz ardı edilir. Ayrıca, işletim sisteminin adres alanlarını ayırt etmek için kullanabileceği 8 bitlik ASID'yi de görüyoruz.

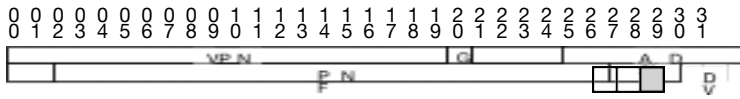


Figure 19.4: **A MIPS TLB Entry**

TIP: RAM ISN'T ALWAYS RAM (CULLER'S LAW)

Rastgele erişim belleği**(random-access memory)** veya RAM terimi, RAM'in herhangi bir bölümüne bir diğeri kadar hızlı erişebileceğiniz anlamına gelir. RAM'i bu şekilde düşünmek genellikle iyi olsa da, TLB gibi donanım/işletim sistemi özellikleri nedeniyle, özellikle o sayfa şu anda TLB'niz tarafından eşlenmemişse, belirli bir bellek sayfasına erişmek maliyetli olabilir. Bu nedenle, uygulama ipucunu hatırlamak her zaman iyidir: RAM her zaman RAM değildir**(RAM isn't always RAM)**. Özellikle erişilen sayfa sayısı TLB kapsamını aşarsa, bazen adres alanınıza rastgele erişim ciddi performans cezalarına neden olabilir. Danışmanlarımızdan biri olan David Culler, birçok performans sorununun kaynağı olarak her zaman TLB'yi gösterdiği için, bu yasayı onun onuruna adlandırıyoruz:**Culler's Law**.

Yukarıda tarif edilen). Size bir soru: Aynı anda çalışan 256'dan (2) fazla işlem varsa işletim sistemi ne yapmalıdır? Son olarak, bir sayfanın donanım tarafından nasıl önbelleğe alınacağını belirleyen 3 Coherence (C) biti görüyoruz.

(bu notların kapsamının biraz ötesinde); sayfa yazıldığında işaretlenen kirli bir bit (bunun kullanımını daha sonra göreceğiz); geçerli bir bit

girişte geçerli bir çeviri olup olmadığını donanıma bildirir. Birden çok sayfa boyutunu destekleyen bir sayfa maskesi alanı da (gösterilmemiştir) vardır; daha büyük sayfalara sahip olmanın neden yararlı olabileceğini ileride göreceğiz. Son olarak, 64 bitin bir kısmı kullanılmamaktadır (şemada gri gölgeli).

MIPS TLB'ler genellikle bu girdilerden 32 veya 64 tanesine sahiptir ve bunların çoğu kullanıcı işlemleri tarafından çalışırken kullanılır. Ancak, birkaçı işletim sistemi için ayrılmıştır. Donanıma işletim sistemi için TLB'nin kaç yuvasının ayrılacağını söylemek için işletim sistemi tarafından kablolu bir kayıt ayarlanabilir; işletim sistemi bu ayrılmış eşlemeleri, kritik zamanlarda erişmek istediği kod ve veriler için kullanır; burada bir TLB kaçırmanın sorunlu olacağı durumlarda (örneğin, TLB kaçırma işleyicisinde).

MIPS TLB yazılım tarafından yönetildiğinden, TLB'yi güncellemek için talimatların olması gerekir. MIPS bu tür dört talimat sağlar: TLB'yi belirli bir çevirinin orada olup olmadığını görmek için araştıran TLBP; Kayıtlara bir TLB girişinin içeriğini okuyan TLBR; Belirli bir TLB girişinin yerini alan TLBWI; ve rastgele bir TLB girişinin yerini alan TLBWR. İşletim sistemi, TLB'nin içeriğini yönetmek için bu talimatları kullanır. Bu talimatların ayrıcalıklı**(privileged**) olması elbette önemlidir; TLB'nin içeriğini değiştirebilseydi bir kullanıcı işleminin neler yapabileceğini hayal edin (ipucu: hemen hemen her şey, makineyi ele geçirmek, kendi kötü niyetli "işletim sistemini" çalıştırmak ve hatta Sun'ı ortadan kaldırmak dahil).

19.8 Summary

Donanımın adres çevirisini daha hızlı yapmamıza nasıl yardımcı olabileceğini gördük. Bir adres çeviri önbelleği olarak küçük, özel bir çip üzerinde TLB sağlayarak, çoğu bellek referansı umarız
ana bellekteki sayfa tablosuna erişmek için. Böylece, ortak davada,

programın performansı neredeyse hiç sanallaştırılmamış gibi olacak, bu bir işletim sistemi için mükemmel bir başarı ve kesinlikle modern sistemlerde sayfalama kullanımı için gerekli.

Ancak, TLB'ler var olan her program için dünyayı pembeleştirmez. Özellikle, bir programın kısa sürede eriştiği sayfa sayısı, TLB'ye sığan sayfa sayısını aşarsa, program çok sayıda TLB hatası üretecek ve bu nedenle oldukça yavaş çalışacaktır. Bu fenomeni TLB kapsamını aşmak**(TLB cov- erage)** olarak adlandırıyoruz, programın performansı neredeyse hiç sanallaştırılmamış gibi olacak, bu bir işletim sistemi için mükemmel bir başarı ve kesinlikle modern sistemlerde sayfalama kullanımı için gerekli.

ve bazı programlar için oldukça sorun olabilir. Bir sonraki bölümde tartışacağımız gibi bir çözüm, daha büyük sayfa boyutları için destek eklemektir; anahtar veri yapılarını programın adres alanının daha büyük sayfalarla eşlenen bölgelerine eşleyerek, TLB'nin etkin kapsamı artırılabilir. Büyük sayfalara yönelik destek, hem büyük hem de rastgele erişilen belirli veri yapılarına sahip bir veritabanı yönetim sistemi**(database management system)** (DBMS) gibi programlar tarafından sıklıkla istismar edilir.

Bahsetmeye değer başka bir TLB sorunu: TLB erişimi, özellikle fiziksel olarak indekslenmiş önbellek**(physically-indexed cache)** olarak adlandırılan şeyle, CPU ardışık düzeninde kolayca bir darboğaza dönüşebilir. Böyle bir önbellekte, önbelleğe erişilmeden önce adres çevirisinin yapılması gerekir, bu da işleri biraz yavaşlatabilir. Bu olası sorun nedeniyle, insanlar sanal adreslerle önbelleklere **(virtually- indexed cache)** erişmenin her türlü akıllı yolunu aradılar ve böylece bir önbellek isabeti durumunda pahalı çeviri adımından kaçındılar. Bu tür bir sanal dizinlenmiş önbellek, bazı performans sorunlarını çözer, ancak donanım tasarımına da yeni sorunlar getirir. Daha fazla ayrıntı için Wiggins'in ince anketine bakın [W03].

References

[BC91] "Performance from Architecture: Comparing a RISC and a CISC with Similar Hard- ware Organization" by D. Bhandarkar and Douglas W. Clark. Communications of the ACM, September 1991. *A great and fair comparison between RISC and CISC. The bottom line: on similar hardware, RISC was about a factor of three better in performance.*

[CM00] "The evolution of RISC technology at IBM" by John Cocke, V. Markstein. IBM Journal of Research and Development, 44:1/2. *A summary of the ideas and work behind the IBM 801, which many consider the first true RISC microprocessor.*

[C95] "The Core of the Black Canyon Computer Corporation" by John Couleur. IEEE Annals of History of Computing, 17:4, 1995. *In this fascinating historical note, Couleur talks about how he invented the TLB in 1964 while working for GE, and the fortuitous collaboration that thus ensued with the Project MAC folks at MIT.*

[CG68] "Shared-access Data Processing System" by John F. Couleur, Edward L. Glaser. Patent 3412382, November 1968. *The patent that contains the idea for an associative memory to store address translations. The idea, according to Couleur, came in 1964.*

[CP78] "The architecture of the IBM System/370" by R.P. Case, A. Padegs. Communications of the ACM. 21:1, 73-96, January 1978. *Perhaps the first paper to use the term* ***translation lookaside buffer****. The name arises from the historical name for a cache, which was a* ***lookaside buffer*** *as called by those developing the Atlas system at the University of Manchester; a cache of address translations thus became a* ***translation lookaside buffer****. Even though the term lookaside buffer fell out of favor, TLB seems to have stuck, for whatever reason.*

[H93] "MIPS R4000 Microprocessor User's Manual". by Joe Heinrich. Prentice-Hall, June 1993. Available: http://cag.csail.mit.edu/raw/ . documents/R4400 Uman book Ed2.pdf *A manual, one that is surprisingly readable. Or is it?*

[HP06] "Computer Architecture: A Quantitative Approach" by John Hennessy and David Pat- terson. Morgan-Kaufmann, 2006. *A great book about computer architecture. We have a particular attachment to the classic first edition.*

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

[PS81] "RISC-I: A Reduced Instruction Set VLSI Computer" by D.A. Patterson and C.H. Se- quin. ISCA '81, Minneapolis, May 1981. *The paper that introduced the term RISC, and started the avalanche of research into simplifying computer chips for performance.*

[SB92] "CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking" by Rafael H. Saavedra-Barrera. EECS Department, University of California, Berkeley. Technical Report No. UCB/CSD-92-684, February 1992.. *A great dissertation about how to predict execution time of applications by breaking them down into constituent pieces and knowing the cost of each piece. Probably the most interesting part that comes out of this work is the tool to measure details of the cache hierarchy (described in Chapter 5). Make sure to check out the wonderful diagrams therein.*

[W03] "A Survey on the Interaction Between Caching, Translation and Protection" by Adam Wiggins. University of New South Wales TR UNSW-CSE-TR-0321, August, 2003. *An excellent survey of how TLBs interact with other parts of the CPU pipeline, namely hardware caches.*

[WG00] "The SPARC Architecture Manual: Version 9" by David L. Weaver and Tom Germond. SPARC International, San Jose, California, September 2000. Available: `www.sparc.org/standards/SPARCV9.pdf`. *Another manual. I bet you were hoping for a more fun citation to end this chapter.*

Homework (Measurement)

Bu ödevde, bir TLB'ye erişimin boyutunu ve maliyetini ölçeceksiniz. Bu fikir, tümü çok basit bir kullanıcı düzeyinde programla önbellek hiyerarşilerinin çeşitli yönlerini ölçmek için basit ama güzel bir yöntem geliştiren Saavedra-Barrera'nın [SB92] çalışmasına dayanmaktadır. Daha fazla ayrıntı için çalışmalarını okuyun.

Temel fikir, büyük bir veri yapısı (örneğin bir dizi) içindeki bazı sayfalara erişmek ve bu erişimleri zamanlamaktır. Örneğin, bir makinenin TLB boyutunun 4 olduğunu varsayalım (ki bu çok küçük ama bu tartışmanın amaçları açısından yararlı olacaktır). 4 veya daha az sayfaya dokunan bir program yazarsanız, her erişim bir TLB isabeti olmalı ve bu nedenle nispeten hızlı olmalıdır. Bununla birlikte, bir döngüde art arda 5 veya daha fazla sayfaya dokunduğunuzda, her erişimin maliyeti birdenbire bir TLB'nin kaçırılmasına neden olacaktır.

Bir dizide bir kez döngü yapmak için temel kod şöyle görünmelidir:

```
int jump = PAGESIZE / sizeof(int);
for (i = 0; i < NUMPAGES * jump; i += jump)
    a[i] += 1;
```

Bu döngüde, NUMPAGES tarafından belirtilen sayfa sayısına kadar a dizisinin her sayfası için bir tamsayı güncellenir. Böyle bir döngüyü art arda zamanlayarak (diyelim ki, bunun etrafındaki başka bir döngüde birkaç yüz milyon kez veya birkaç saniye çalışması için birçok döngüye ihtiyaç duyulursa), her bir erişimin (ortalama olarak) ne kadar sürdüğünü ölçebilirsiniz. NUMPAGES arttıkça maliyetteki sıçramaları arayarak, kabaca birinci düzey TLB'nin ne kadar büyük olduğunu belirleyebilir, ikinci düzey TLB'nin var olup olmadığını (ve varsa ne kadar büyük olduğunu) belirleyebilirsiniz ve genel olarak bu konuda iyi bir fikir edinebilirsiniz. TLB isabetlerinin ve ıskalamalarının performansı nasıl etkileyebileceğini.

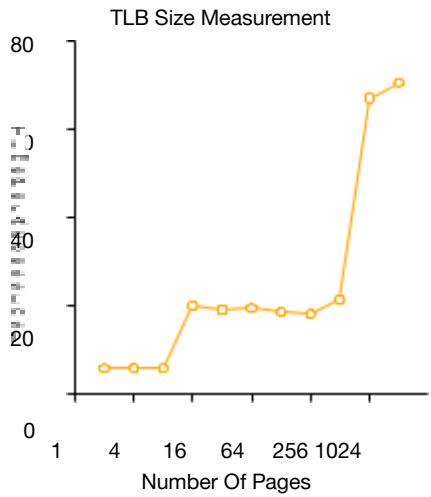


Figure 19.5: **Discovering TLB Sizes and Miss Costs**

Şekil 19.5 (sayfa 15), döngüde erişilen sayfa sayısı arttıkça erişim başına ortalama süreyi göstermektedir. Grafikte görebileceğiniz gibi, yalnızca birkaç sayfaya erişildiğinde (8 veya daha az), ortalama erişim süresi kabaca 5 nanosaniyedir. 16 veya daha fazla sayfaya erişildiğinde, erişim başına yaklaşık 20 nanosaniyeye ani bir sıçrama olur. Maliyette son bir sıçrama yaklaşık 1024 sayfada gerçekleşir ve bu noktada her erişim yaklaşık 70 nanosaniye sürer. Bu verilerden, iki seviyeli bir TLB hiyerarşisi olduğu sonucuna varabiliriz; ilki oldukça küçüktür (muhtemelen 8 ila 16 giriş tutar); ikincisi daha büyük ama daha yavaştır (yaklaşık 512 giriş tutar). Birinci seviye TLB'deki isabetler ile ıskalamalar arasındaki genel fark oldukça büyüktür, kabaca on dört kattır. TLB performansı önemlidir!

Questions

1. Zamanlama için bir zamanlayıcı kullanmanız gerekir (ör. gettimeofday()). Böyle bir zamanlayıcı ne kadar hassastır? Tam olarak zamanlamanız için bir operasyonun ne kadar sürmesi gerekir? (bu, başarılı bir şekilde zamanlamak için bir sayfa erişimini bir döngüde kaç kez tekrarlamanız gerekeceğini belirlemenize yardımcı olacaktır)

CLOCK_PROCESS_CPUTIME_ID, Linux'ta 1 nanosaniye, macOS'ta 1000 nanosaniye çözünürlüğe sahiptir.

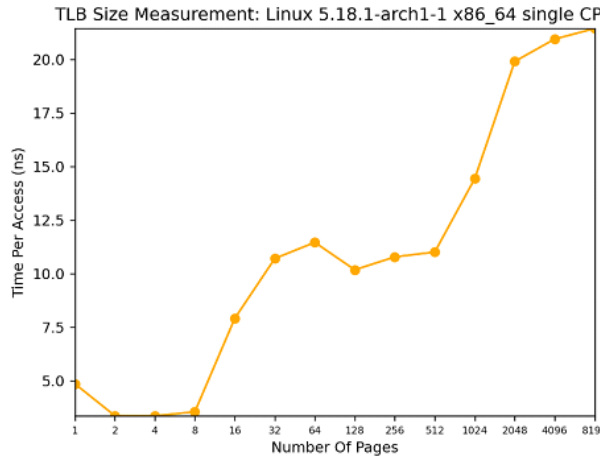
2. Her sayfaya erişim maliyetini kabaca ölçebilen tlb.c adlı programı yazın. Programın girdileri şunlar olmalıdır: dokunulacak sayfa sayısı ve deneme sayısı.

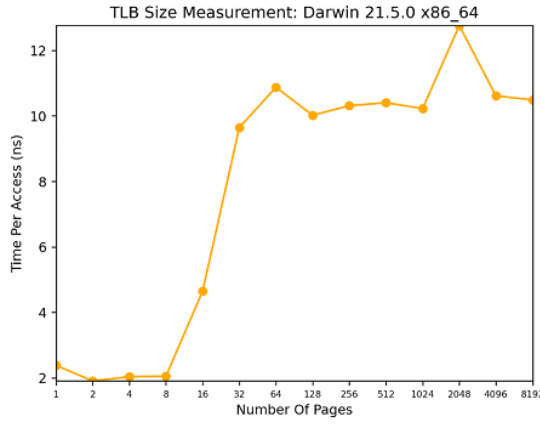
3. Şimdi, erişilen sayfa sayısını 1'den birkaç bine kadar değiştirerek, belki yineleme başına iki kat artırarak, bu programı çalıştırmak için en sevdiğiniz betik dilinde (bash?) bir betik yazın. Komut dosyasını farklı makinelerde çalıştırın ve bazı veriler toplayın. Güvenilir ölçümler elde etmek için kaç deneme gerekiyor?

100000 deneme.

4. Ardından, yukarıdakine benzer görünen bir grafik oluşturarak sonuçları çizin. Ploticus ve hatta zplot gibi iyi bir araç kullanın. Görselleştirme genellikle verileri sindirmeyi çok daha kolaylaştırır; Neden bu olduğunu düşünüyorsun?

```
$ make
$ python plot.py 14 100000 --single_cpu
```





5. Dikkat edilmesi gereken bir şey derleyici optimizasyonudur. Derleyiciler, programın başka hiçbir bölümünün kullanmadığı değerleri artıran döngüleri kaldırmak da dahil olmak üzere her türlü zekice şeyi yapar. Derleyicinin yukarıdaki ana döngüyü TLB boyut tahmincinizden çıkarmamasını nasıl sağlayabilirsiniz?

Optimizasyonu devre dışı bırakmak için gcc'nin optimize etme seçeneği gcc -O0'ı kullanma. Bu varsayılan ayardır.

6. Dikkat edilmesi gereken başka bir şey de, günümüzde çoğu sistemin birden fazla CPU ile gönderildiği ve elbette her CPU'nun kendi TLB

hiyerarşisine sahip olduğu gerçeğidir. Gerçekten iyi ölçümler elde etmek için, programlayıcının kodu bir CPU'dan diğerine atlamasına izin vermek yerine, kodunuzu yalnızca bir CPU'da çalıştırmanız gerekir. Nasıl yaparsın? (ipucu: bazı ipuçları için Google'da "pinning a thread" konusuna bakın) Bunu yapmazsanız ve kod bir CPU'dan diğerine geçerse ne olur?

Linux'ta sched_setaffinity(2), pthread_setaffinity_np(3), tasket(1) veya sudo systemd-run -p AllowedCPUs=0 ./tlb.out, FreeBSD'de cpuset_setaffinity(2) veya cpuset(1) kullanın.
Veya hwloc-bind package:0.pu:0 -- ./tlb.out kullanın.

7. Ortaya çıkabilecek başka bir sorun başlatma ile ilgilidir. Diziye erişmeden önce yukarıdaki a'yı başlatmazsanız, talebin sıfırlanması gibi ilk erişim maliyetleri nedeniyle diziye ilk kez eriştiğinizde çok pahalı olacaktır. Bu, kodunuzu ve zamanlamasını etkiler mi? Bu potansiyel maliyetleri dengelemek için ne yapabilirsiniz?
Diziyi başlatmak için calloc(3) kullanın ve ardından zamanı ölçün.