21

Fiziksel Belleğin Ötesinde: Mekanizmalar (Beyond Physical Memory: Mechanisms)

Şimdiye kadar, bir adres alanının gerçekçi olmayan şekilde küçük olduğunu ve fiziksel belleğe sığdığını varsaydık. Aslında, çalışan her işlemin her adres alanının belleğe uyduğunu varsayıyoruz. Şimdi bu büyük varsayımları gevşeteceğiz ve aynı anda çalışan birçok büyük adres alanını desteklemek istediğimizi varsayacağız.

Bunu yapmak için, **bellek hiyerarşisinde (memory hierarchy)** ek bir seviyeye ihtiyacımız var. Şimdiye kadar, tüm sayfaların fiziksel bellekte bulunduğunu varsaydık. Bununla birlikte, büyük adres alanlarını desteklemek için işletim sisteminin, şu anda büyük ölçüde kullanılmayan adres alanlarının bölümlerini saklayacak bir yere ihtiyacı olacaktır. Genel olarak, böyle bir konumun özellikleri, bellekten daha fazla kapasiteye sahip olması gerektiğidir; Sonuç olarak, genellikle daha yavaştır (daha hızlı olsaydı, onu yalnızca bellek olarak kullanırdık, değil mi?). Modern sistemlerde bu role genellikle bir **sabit disk sürücüsü (hard disk drive)** hizmet eder. Bu nedenle, bellek hiyerarşimizde, büyük ve yavaş sabit diskler en altta, bellek hemen üstte yer alır. Ve böylece sorunun can alıcı noktasına geliyoruz:

KRİTİK NOKTA: FİZİKSEL BELLEĞİN ÖTESİNE NASIL GİDİLİR?
İşletim sistemi, büyük bir sanal adres alanı yanılsamasını şeffaf bir şekilde sağlamak için daha büyük, daha yavaş bir aygıtı nasıl kullanabilir?

Aklınıza gelebilecek bir soru: bir işlem için neden tek bir büyük adres alanını desteklemek istiyoruz? Cevap bir kez daha rahatlık ve kullanım kolaylığıdır. Geniş bir adres alanıyla, programınızın veri yapıları için bellekte yeterli yer olup olmadığı konusunda endişelenmenize gerek yoktur; bunun yerine, programı doğal bir şekilde yazarsınız ve gerektiğinde bellek ayırırsınız. İşletim sisteminin sağladığı güçlü bir yanılsamadır ve hayatınızı büyük ölçüde kolaylaştırır. Önemli değil! Programcıların kod veya veri parçalarını gerektiği gibi belleğe girip çıkarmasını gerektiren **bellek bindirmelerini (memory overlays)** kullanan eski sistemlerde bir karşıtlık bulunur [D97]. Bunun nasıl olacağını hayal etmeye çalışın: bir işlevi çağırmadan veya bazı verilere erişmeden önce, önce kodun veya verilerin bellekte olmasını ayarlamanız gerekir; iğrenç!

KENAR NOT: DEPOLAMA TEKNOLOJİLERİ
G / Ç cihazlarının gerçekte nasıl çalıştığına daha sonra daha derinlemesine bakacağız (G / Ç cihazlarıyla ilgili bölüme bakın). Bu yüzden sabırlı ol! Ve tabii ki daha yavaş aygıtın bir sabit disk olması gerekmez, ancak Flash tabanlı bir SSD (katı hal sürücüsü) gibi daha modern bir şey olabilir. O konulardan da bahsedeceğiz. Şimdilik, fiziksel belleğin kendisinden bile daha büyük, çok büyük bir sanal bellek yanılsaması oluşturmamıza yardımcı olması için kullanabileceğimiz büyük ve nispeten yavaş bir cihazımız olduğunu varsayalım.

Tek bir işlemin ötesinde, takas alanı eklenmesi, işletim sisteminin aynı anda çalışan birden çok işlem için büyük bir sanal bellek yanılsamasını desteklemesine olanak tanır. Çoklu programlamanın icadı (makineyi daha iyi kullanmak için birden çok programı "hepsini birlikte" çalıştırmak) neredeyse bazı sayfaların yer değiştirmesini gerektiriyordu, çünkü eski makineler açıkça tüm süreçlerin ihtiyaç duyduğu tüm sayfaları aynı anda tutamazdı. Bu nedenle, çoklu programlama ve kullanım kolaylığının birleşimi, fiziksel olarak mevcut olandan daha fazla bellek kullanmayı desteklemek istememize neden olur. Bu, tüm modern sanal makine sistemlerinin yaptığı bir şeydir; şimdi hakkında daha fazla şey öğreneceğimiz bir şey.

21.1 Takas Alanı

Yapmamız gereken ilk şey, sayfaları ileri geri taşımak için diskte biraz yer ayırmaktır. İşletim sistemlerinde, genellikle **takas alanı (swap space)** gibi bir alana atıfta bulunuruz, çünkü sayfaları bellekten ona değiştiririz ve sayfaları bellekten belleğe değiştiririz. Bu nedenle, işletim sisteminin sayfa boyutlu birimlerde takas alanını okuyabildiğini ve takas alanına yazabildiğini varsayacağız. Bunu yapmak için işletim sisteminin belirli bir sayfanın **disk adresini (disk address)** hatırlaması gerekir.

Takas alanının boyutu önemlidir, çünkü sonuçta bir sistem tarafından belirli bir zamanda kullanılabilecek maksimum bellek sayfası sayısını belirler. Basitlik için şimdilik çok büyük olduğunu varsayalım.

Küçük örnekte (Şekil 21.1), 4 sayfalık bir fiziksel bellek ve 8 sayfalık bir takas alanının küçük bir örneğini görebilirsiniz. Örnekte, üç işlem (Proc 0, Proc 1 ve Proc 2) fiziksel verileri etkin bir şekilde paylaşmaktadır; Ancak üçünün her birinin geçerli sayfalarının yalnızca bir kısmı bellekte, geri kalanı diskteki takas alanında bulunur. Dördüncü bir işlem (Proc 3), tüm sayfalarını diske kaydırdı ve bu nedenle şu anda açıkça çalışmıyor. Bir takas bloğu ücretsiz kalır. Bu küçük örnekten bile, umarım takas alanını kullanmanın sistemin belleğin gerçekte olduğundan daha büyük olduğunu iddia etmesine nasıl izin verdiğini görebilirsiniz.

Takas alanının, takas trafiği için tek disk konumu olmadığını unutmamalıyız. Örneğin, ikili bir program çalıştırdığınızı varsayalım (ör. ls veya kendi derlenmiş ana programınız). Bu ikili dosyanın kod sayfaları başlangıçta diskte bulunur ve program çalıştığında belleğe yüklenir (program çalışmaya başladığında tümü birden

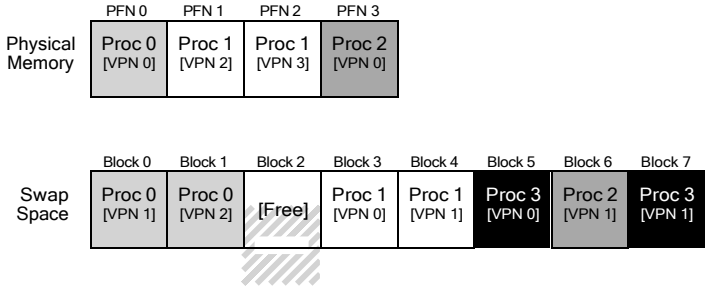

Figure 21.1: **Physical Memory and Swap Space**

veya modern sistemlerde olduğu gibi, gerektiğinde her seferinde bir sayfa). Bununla birlikte, sistemin diğer ihtiyaçlar için fiziksel bellekte yer açması gerekirse, daha sonra dosya sistemindeki disk üzerindeki ikili dosyadan yeniden değiştirebileceğini bilerek, bu kod sayfaları için bellek alanını güvenli bir şekilde yeniden kullanabilir.

21.2 Mevcut Bit

Artık diskte biraz yerimiz olduğuna göre, diske ve diskten sayfa değiştirmeyi desteklemek için sisteme daha yükseğe bazı makineler eklememiz gerekiyor. Basit olması için, donanım tarafından yönetilen bir TLB'ye sahip bir sistemimiz olduğunu varsayalım.

Önce bir bellek referansında ne olduğunu hatırlayın. Çalışan işlem sanal bellek başvuruları oluşturur (yönerge getirmeleri veya veri erişimleri için) ve bu durumda donanım, istenen verileri bellekten getirmeden önce bunları fiziksel adreslere çevirir.

Donanımın önce VPN'i (sanal özel ağ) sanal adresten çıkardığını, TLB ile bir eşleşme olup olmadığını **(TLB isabeti) (TLB hit)** kontrol ettiğini ve bir isabet olursa ortaya çıkan fiziksel adresi üretip bellekten getirdiğini unutmayın. Bu, hızlı olduğu için (ek bellek erişimi gerektirmediğinden) umarım yaygın bir durumdur. VPN tlb'de bulunmazsa (yani bir **TLB eksikliği) (TLB miss)**, donanım sayfa tablosunu bellekte bulur (**sayfa tablosu temel kaydını** kullanarak) **(page table base register)** ve vpn'yi dizin olarak kullanarak bu sayfa için **sayfa tablosu girişini** (PTE) **(page table entry)** arar. Sayfa geçerliyse ve fiziksel bellekte bulunuyorsa, donanım PFN'yi PTE'den çıkarır, TLB'ye yükler ve yeniden dener. talimat, bu kez bir TLB isabeti oluşturuyor; Şimdiye kadar, çok iyi.

Bununla birlikte, sayfaların diske değiştirilmesine izin vermek istiyorsak, daha da fazla makine eklemeliyiz. Özellikle, donanım PTE'YE baktığında, sayfanın fiziksel bellekte bulunmadığını görebilir. Donanımın (veya yazılımla yönetilen TLB yaklaşımında işletim sisteminin) bunu belirleme yolu, her sayfa tablosu girişindeki **mevcut bit (present bit)** olarak bilinen yeni bir bilgi parçasıdır. Mevcut bit bir olarak ayarlanmışsa, bu, sayfanın fiziksel bellekte olduğu ve her şeyin yukarıdaki gibi ilerlediği anlamına gelir; sıfıra ayarlanırsa, sayfa bellekte değil, diskte bir yerdedir. Fiziksel bellekte olmayan bir sayfaya erişme eylemi genellikle **sayfa hatası (page fault)** olarak adlandırılır.

BİR KENARA: TERMİNOLOJİ VE DİĞER ŞEYLERİ DEĞİŞTİRME
Sanal bellek sistemlerindeki terminoloji, makineler ve işletim sistemleri arasında biraz kafa karıştırıcı ve değişken olabilir. Örneğin, bir **sayfa hatası (page fault)** daha genel olarak, bir tür hata oluşturan bir sayfa tablosuna yapılan herhangi bir referansa atıfta bulunabilir: bu, burada tartıştığımız hata türünü, yani sayfa yok hatası içerebilir, ancak bazen yasa dışı bellek erişimlerine bakın. Gerçekten de, kesinlikle yasal bir erişim olanı (bir sürecin sanal adres alanına eşlenen, ancak o sırada fiziksel bellekte olmayan bir sayfaya) "hata" olarak adlandırmamız gariptir; Gerçekten, **sayfa özlemesi (page miss)** olarak adlandırılmalıdır. Ancak çoğu zaman, insanlar bir programın "sayfa hatası" olduğunu söylediğinde, işletim sisteminin diske değiştirdiği sanal adres alanının bölümlerine eriştiği anlamına gelir.

Bu davranışın bir "hata" olarak bilinmeye başlanmasının nedeninin, işletim sistemindeki bununla başa çıkmak için kullanılan mekanizmayla ilgili olduğundan şüpheleniyoruz. Alışılmadık bir şey olduğunda, yani donanımın nasıl başa çıkacağını bilmediği bir şey olduğunda, donanım, işleri daha iyi hale getirebileceğini umarak kontrolü basitçe işletim sistemine aktarır. Bu durumda, bir işlemin erişmek istediği bir sayfa bellekte eksiktir; donanım yapabileceği tek şeyi yapar, bu da bir istisna oluşturur ve işletim sistemi oradan devralır. Bu, bir süreç yasa dışı bir şey yaptığında olanla aynı

Bir sayfa hatası üzerine, işletim sistemi sayfa hatasına hizmet vermek için çağrılır. **Sayfa hatası işleyicisi (page-fault handler)** olarak bilinen belirli bir kod parçası çalışır ve şimdi açıkladığımız gibi sayfa hatasına hizmet etmelidir.

21.3 Sayfa Hatası İşleyici

TLB kayıplarında iki tür sistemimiz olduğunu hatırlayın: donanım tarafından yönetilen TLB'ler (donanımın istenen çeviriyi bulmak için sayfa tablosuna baktığı yer) ve yazılım tarafından yönetilen TLB'ler (işletim sisteminin yaptığı yer). Her iki sistem türünde de, bir sayfa yoksa, sayfa hatasını işlemek için işletim sistemi görevlendirilir. Uygun şekilde adlandırılmış işletim sistemi **sayfa hatası işleyicisi (page-fault handler)**, ne yapılacağını belirlemek için çalışır. Neredeyse tüm sistemler yazılımdaki sayfa hatalarını işler; donanım tarafından yönetilen bir TLB ile bile donanım, bu önemli görevi yönetmesi için işletim sistemine güvenir.

Bir sayfa mevcut değilse ve diske değiştirilmişse, işletim sisteminin sayfa hatasını gidermek için sayfayı belleğe değiştirmesi gerekir. Böylece bir soru ortaya çıkıyor: İşletim sistemi istenen sayfayı nerede bulacağını nasıl bilecek? Birçok sistemde, sayfa tablosu bu tür bilgileri depolamak için doğal bir yerdir. Böylece işletim sistemi, bir disk adresi için sayfanın PFN'si gibi veriler için normalde kullanılan PTE'deki bitleri kullanabilir. İşletim Sistemi bir sayfa için bir sayfa hatası aldığında, adresi bulmak için PTE'ye bakar ve sayfayı belleğe almak için diske istek gönderir.

BİR KENARA: DONANIM NEDEN SAYFA HATALARINI ELE ALMIYOR

TLB ile olan deneyimlerimizden, donanım tasarımcılarının pek çok şeyi yapması için işletim sistemine güvenmekten nefret ettiklerini biliyoruz. Peki neden bir sayfa hatasıyla başa çıkmak için işletim sistemine güveniyorlar? Bunun birkaç ana nedeni var. İlk olarak, diskteki sayfa hataları yavaştır; İşletim sisteminin tonlarca talimatı yürüterek bir hatayı halletmesi uzun zaman alsa bile, disk işleminin kendisi geleneksel olarak o kadar yavaştır ki, çalışan yazılımın ekstra genel giderleri minimum düzeydedir. İkincisi, bir sayfa hatasını işleyebilmek için, donanımın takas alanını, diske G/Ç'lerin nasıl verileceğini ve şu anda hakkında fazla bir şey bilmediği birçok başka ayrıntıyı anlaması gerekir. Bu nedenle, hem

Disk G / Ç tamamlandığında, işletim sistemi daha sonra sayfayı mevcut olarak işaretlemek için sayfa tablosunu günceller, yeni getirilen sayfanın bellek içi konumunu kaydetmek için sayfa tablosu girişinin (PTE) PFN alanını günceller ve talimatı yeniden dener. Bu sonraki girişim, daha sonra hizmet verilecek ve TLB'yi çeviri ile güncelleyecek bir TLB eksikliği oluşturabilir (bu adımdan kaçınmak için sayfa hatasına hizmet verilirken dönüşümlü olarak TLB güncellenebilir). Son olarak, son bir yeniden başlatma, çeviriyi TLB'de bulur ve böylece istenen verileri veya yönergeyi, çevrilen fiziksel adreste bellekten almaya devam eder.

G/Ç hareket halindeyken, işlemin **engellenmiş (blocked)** durumda olacağını unutmayın. Böylece, sayfa hatasına hizmet verilirken işletim sistemi diğer hazır işlemleri çalıştırmakta özgür olacaktır. G/Ç pahalı olduğundan, bir işlemin G/Ç'si (sayfa hatası) ile diğerinin yürütülmesi arasındaki bu **çakışma (overlap)**, çok programlı bir sistemin donanımını en etkili şekilde kullanmasının başka bir yoludur.

21.4 Bellek Doluysa Ne Olur?

Yukarıda açıklanan işlemde, takas alanından bir sayfada **sayfa (page in)** açmak için bol miktarda boş bellek olduğunu varsaydığımızı fark edebilirsiniz. Elbette durum böyle olmayabilir; bellek dolu (veya ona yakın) olabilir. Bu nedenle, İşletim Sistemi, işletim sisteminin getirmek üzere olduğu yeni sayfalara yer açmak için önce bir veya daha fazla **sayfayı çıkarmak (page out)** isteyebilir. Atılacak veya değiştirilecek bir sayfa seçme işlemi, **sayfa değiştirme ilkesi (page-replacement policy)** olarak bilinir.

Görünüşe göre, yanlış sayfanın atılması program performansında büyük bir maliyete neden olabileceğinden, iyi bir sayfa değiştirme politikası oluşturmaya çok fazla düşünce harcanmıştır. Yanlış karar vermek, bir programın bellek benzeri hızlar yerine disk benzeri hızlarda çalışmasına neden olabilir; mevcut teknolojide bu, bir programın 10.000 veya 100.000 kat daha yavaş çalışabileceği anlamına gelir. Bu nedenle, böyle bir politika biraz ayrıntılı olarak incelememiz gereken bir şeydir; Aslında, bir sonraki bölümde yapacağımız şey tam olarak budur. Şimdilik, burada açıklanan mekanizmaların üzerine inşa edilmiş böyle bir politikanın var olduğunu anlamak yeterince iyi.

```
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
16         if (CanAccess(PTE.ProtectBits) == False)
17             RaiseException(PROTECTION_FAULT)
18         else if (PTE.Present == True)
19             // assuming hardware-managed TLB
20             TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
21             RetryInstruction()
22         else if (PTE.Present == False)
23             RaiseException(PAGE_FAULT)
```

Şekil 21.2: **Sayfa Hatası Kontrol Akış Algoritması (Donanım)**

21.5 Sayfa Hata Kontrol Akışı

Tüm bu bilgiler yerinde olduğunda, artık bellek erişiminin tam kontrol akışını kabaca çizebiliriz. Başka bir deyişle, biri size "bir program bellekten bazı veriler getirdiğinde ne olur?" diye sorduğunda, tüm farklı olasılıklar hakkında oldukça iyi bir fikriniz olmalıdır. Daha fazla ayrıntı için Şekil 21.2 ve 21.3'teki kontrol akışına bakın; ilk şekil çeviri sırasında donanımın ne yaptığını, ikincisi ise işletim sisteminin bir sayfa hatası olduğunda ne yaptığını gösterir.

Şekil 21.2'deki donanım kontrol akış şemasından, bir TLB eksikliği oluştuğunda anlaşılması gereken üç önemli durum olduğuna dikkat edin. İlk olarak, sayfanın hem **mevcut (present)** hem de **geçerli (valid)** olduğu (18-21. Satırlar); bu durumda, TLB hata işleyicisi PTE'den PFN'yi kolayca alabilir, talimatı yeniden deneyebilir (bu sefer bir TLB isabetiyle sonuçlanır) ve böylece daha önce açıklandığı gibi (birçok kez) devam edebilir. İkinci durumda (Satır 22–23), sayfa hatası işleyici çalıştırılmalıdır; bu, işlemin erişmesi için meşru bir sayfa olmasına rağmen (sonuçta geçerlidir), fiziksel bellekte mevcut değildir. Üçüncüsü (ve son olarak), örneğin programdaki bir hata nedeniyle erişim geçersiz bir sayfaya olabilir (Satır 13-14). Bu durumda, PTE'deki diğer hiçbir bit gerçekten önemli değildir; donanım bu geçersiz erişimi yakalar ve işletim sistemi tuzak işleyicisi çalışır ve büyük olasılıkla rahatsız edici işlemi sonlandırır.

Şekil 21.3'teki yazılım kontrol akışından, sayfa hatasını onarmak için işletim sisteminin kabaca ne yapması gerektiğini görebiliriz. İlk olarak, işletim sistemi, yakında hatalı olacak sayfanın içinde bulunacağı fiziksel bir çerçeve bulmalıdır; böyle bir sayfa yoksa, değiştirme algoritmasının çalışmasını beklememiz ve bazı sayfaları bellekten atarak burada kullanım için serbest bırakmamız gerekir.

```
1  PFN = FindFreePhysicalPage()
2  if (PFN == -1)              // boş sayfa bulunamadı
3      PFN = EvictPage()       // değiştirme algoritmasını çalıştır
4  DiskRead(PTE.DiskAddr, PFN) // uyku (G/Ç için bekleniyor)
5  PTE.present = True          // mevcut sayfa tablosunu güncelle
6  PTE.PFN     = PFN           // bit ve çeviri (PFN)
7  RetryInstruction()          // talimatı yeniden dene
```

Şekil 21.3: **Sayfa Hatası Kontrol Akış Algoritması (Yazılım)**

Eldeki fiziksel bir çerçeveyle, işleyici daha sonra takas alanından sayfada okumak için G / Ç isteğini yayınlar. Son olarak, bu yavaş işlem tamamlandığında, işletim sistemi sayfa tablosunu günceller ve talimatı yeniden dener. Yeniden deneme, bir TLB hatasıyla sonuçlanacak ve ardından, başka bir yeniden denemede, bir TLB isabetiyle sonuçlanacak ve bu noktada donanım, istenen öğeye erişebilecektir.

21.6 Değiştirmeler Gerçekten Gerçekleştiğinde

Şimdiye kadar, değiştirmelerin nasıl gerçekleştiğini açıklama şeklimiz, işletim sisteminin bellek tamamen dolana kadar beklediğini ve ancak o zaman başka bir sayfaya yer açmak için bir sayfanın yerini aldığını (tahliye ettiğini) varsayar. Tahmin edebileceğiniz gibi, bu biraz gerçekçi değil ve işletim sisteminin belleğin küçük bir bölümünü daha proaktif bir şekilde boş tutmasının birçok nedeni var.

Az miktarda belleği boş tutmak için çoğu işletim sistemi, sayfaları bellekten çıkarmaya ne zaman başlayacağına karar vermeye yardımcı olmak için bir tür **yüksek filigran (HW) (high watermark)** ve **düşük filigran (LW) (low watermark)** içerir. Bunun nasıl çalıştığı şu şekildedir: OS, kullanılabilir LW sayfalarından daha azının olduğunu fark ettiğinde, belleği boşaltmaktan sorumlu olan bir arka plan iş parçacığı çalışır. İş parçacığı, HW sayfaları bulunana kadar sayfaları tahliye eder. Bazen **takas arka plan programı (swap daemon)** veya **sayfa arka plan programı (page daemon[1])** olarak adlandırılan arka plan iş parçacığı, çalışan süreçler ve işletim sisteminin kullanması için belleğin bir kısmını boşalttığı için mutlu olarak uyku moduna geçer.

Aynı anda bir dizi değiştirme gerçekleştirerek, yeni performans optimizasyonları mümkün hale gelir. Örneğin, birçok sistem birkaç sayfayı **kümeler (cluster)** veya **gruplandırır (group)**ve bunları bir kerede takas bölümüne yazar, böylece diskin [LL82] etkinliği artar; Daha sonra diskleri daha ayrıntılı olarak tartıştığımızda göreceğimiz gibi, bu tür bir kümeleme, bir diskin arama ve döndürme ek yüklerini azaltır ve böylece performansı önemli ölçüde artırır.

Arka plan sayfalama iş parçacığıyla çalışmak için Şekildeki kontrol akışı 21.3 biraz değiştirilmelidir; Algoritma doğrudan bir değiştirme yapmak yerine, bunun yerine boş sayfa olup olmadığını kontrol eder. Değilse, arka planda sayfalama iş parçacığına boş sayfalara ihtiyaç duyulduğunu bildirir; İş parçacığı bazı sayfaları boşalttığında, orijinal iş parçacığını yeniden uyandırır, bu da daha sonra istenen sayfada sayfa oluşturabilir ve işine devam edebilir.

Genellikle "iblis" olarak telaffuz edilen "daemon™" kelimesi, yararlı bir şeyler yapan bir arka plan dizisi veya süreci için kullanılan eski bir terimdir. (Bir kez daha!) terimin kaynağının Multics [CS94] olduğu ortaya çıktı.

İPUCU: ARKA PLANDA ÇALIŞIN

Yapacak bazı işleriniz olduğunda, verimliliği artırmak ve operasyonların gruplandırılmasına izin vermek için genellikle bunu **arka planda (background)** yapmak iyi bir fikirdir. İşletim sistemleri genellikle arka planda çalışır; örneğin, birçok sistem arabellek dosyası, verileri diske gerçekten yazmadan önce belleğe yazar. Bunu yapmanın birçok olası faydası vardır: disk artık aynı anda çok sayıda yazma alabileceğinden ve böylece bunları daha iyi zamanlayabildiğinden, artan disk verimliliği; yazma işemlerinin oldukça hızlı tamamlandığını düşündüğü için yazma işlemlerinin gecikme süresi iyileştirildi; yazma işlemlerinin asla diske gitmesi gerekmeyebileceğinden (yani, dosya silinirse) iş azaltma olasılığı; ve **boş zamanın (idle time)** daha iyi kullanılması, çünkü arka plan çalışması muhtemelen sistem boştayken yapılabilir, böylece donanımdan daha iyi yararlanılır [G+95].

21.7 Özet

Bu kısa bölümde, bir sistemde fiziksel olarak mevcut olandan daha fazla belleğe erişme kavramını tanıttık. Bunu yapmak için sayfa tablosu yapılarında daha fazla karmaşıklık gerekir, çünkü sayfanın bellekte olup olmadığını bize bildirmek için mevcut bir bitin (bir tür) dahil edilmesi gerekir. Değilse, işletim sistemi **sayfa hatası işleyicisi (page-fault handler) sayfa hatasına (page fault)** hizmet vermek için çalışır ve böylece istenen sayfanın diskten belleğe aktarımını düzenler, belki de yakında değiştirilecek olanlara yer açmak için önce bellekteki bazı sayfaları değiştirir.

Önemli (ve şaşırtıcı bir şekilde!), bu eylemlerin tümünün süreçte **şeffaf bir şekilde (transparently)** gerçekleştiğini hatırlayın. Süreç söz konusu olduğunda, yalnızca kendi özel, bitişik sanal belleğine erişiyor. Sahne arkasında, sayfalar fiziksel bellekte rastgele (bitişik olmayan) konumlara yerleştirilir ve bazen bellekte bile bulunmazlar, bu da diskten getirmeyi gerektirir. Genel durumda bir bellek erişiminin hızlı olduğunu umarken, bazı durumlarda ona hizmet vermek için birden çok disk işlemi gerekir; tek bir talimatı yerine getirmek kadar basit bir şey, en kötü durumda, tamamlanması milisaniyeler alabilir.

References

[CS94] "Take Our Word For It" by F. Corbato, R. Steinberg. www.takeourword.com/TOW146 (Page 4). *Richard Steinberg writes: "Someone has asked me the origin of the word daemon as it applies to computing. Best I can tell based on my research, the word was first used by people on your team at Project MAC using the IBM 7094 in 1963." Professor Corbato replies: "Our use of the word daemon was inspired by the Maxwell's daemon of physics and thermodynamics (my background is in physics). Maxwell's daemon was an imaginary agent which helped sort molecules of different speeds and worked tirelessly in the background. We fancifully began to use the word daemon to describe background processes which worked tirelessly to perform system chores."*

[D97] "Before Memory Was Virtual" by Peter Denning. In the Beginning: Recollections of Software Pioneers, Wiley, November 1997. *An excellent historical piece by one of the pioneers of virtual memory and working sets.*

[G+95] "Idleness is not sloth" by Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, John Wilkes. USENIX ATC '95, New Orleans, Louisiana. *A fun and easy-to-read discussion of how idle time can be better used in systems, with lots of good examples.*

[LL82] "Virtual Memory Management in the VAX/VMS Operating System" by Hank Levy, P. Lipman. IEEE Computer, Vol. 15, No. 3, March 1982. *Not the first place where page clustering was used, but a clear and simple explanation of how such a mechanism works. We sure cite this paper a lot!*

Ödev (Ölçme)

Bu ev ödevi size yeni bir araç olan vmstat'ı ve bunun bellek, CPU (Merkezi işlemci ünitesi) ve G/Ç kullanımını anlamak için nasıl kullanılabileceğini tanıtır. İlgili README'yi okuyun ve aşağıdaki alıştırmalara ve sorulara geçmeden önce mem.c'deki kodu inceleyin.

Sorular

1. İlk olarak, aynı makineye iki ayrı terminal bağlantısı açın, böylece bir pencerede ve diğerinde bir şeyi kolayca çalıştırabilirsiniz.

   Şimdi, bir pencerede, her saniye makine kullanımıyla ilgili istatistikleri gösteren vmstat 1'i çalıştırın. Çıktısını anlayabilmek için kılavuz sayfasını, ilişkili README'yi ve ihtiyacınız olan diğer bilgileri okuyun. Aşağıdaki alıştırmaların geri kalanı için bu pencereyi vmstat çalışır durumda bırakın.

   Şimdi mem.c programını çok az bellek kullanımıyla çalıştıracağız. Bu, ./mem 1 (yalnızca 1 MB bellek kullanan) yazarak gerçekleştirilebilir. Mem çalıştırılırken CPU kullanım istatistikleri nasıl değişir? Kullanıcı zamanı sütunundaki sayılar mantıklı mı? Aynı anda birden fazla mem örneğini çalıştırırken bu nasıl değişir?

   Bu değer, uygulamanın çalıştırılma süresi boyunca değişebilir. CPU kullanım istatistikleri başlangıçta düşük olabilir, ancak uygulamanın çalışması sırasında CPU kullanım istatistikleri artabilir. Bu, uygulamanın işlem gereksinimlerinin değiştiği anlamına gelebilir. CPU kullanım istatistikleri sürekli değişebilir.

   CPU'nun daha fazla çalıştırdığı için daha fazla bellek ve diğer sistem kaynaklarına ihtiyacı olacaktır, bu da CPU kullanım istatistiklerinde değişikliklere yol açabilir. Eğer sisteminizde yeterli kaynak varsa ve CPU'nun yeterli işlemci gücüne sahipse, birden fazla mem örneğini aynı anda çalıştırmak CPU kullanım istatistiklerini olumlu yönde etkileyebilir. Ancak, eğer sistem kaynakları yetersizse veya CPU yeterli işlemci gücüne sahip değilse, birden fazla mem örneğini aynı anda çalıştırmak CPU kullanım istatistiklerini olumsuz yönde etkileyebilir.

2. Şimdi mem'i çalıştırırken bazı bellek istatistiklerine bakmaya başlayalım. İki sütuna odaklanacağız: swpd (kullanılan sanal bellek miktarı) ve boş (boş bellek miktarı). ./mem 1024'ü (1024 MB ayırır) çalıştırın ve bu değerlerin nasıl değiştiğini izleyin. Ardından çalışan programı sonlandırın (control-c yazarak) ve değerlerin nasıl değiştiğini tekrar izleyin. Değerler hakkında ne fark ediyorsunuz? Özellikle, program çıktığında boş sütun nasıl değişir? Mem çıktığında boş bellek miktarı beklenen miktarda artar mı?
3. Ardından, diske ve diskten ne kadar takas yapıldığını gösteren takas sütunlarına (sı ve benzeri) bakacağız. Tabii ki, bunları etkinleştirmek için mem'i büyük miktarda bellekle çalıştırmanız gerekecek. İlk olarak, Linux sisteminizde ne kadar boş bellek olduğunu inceleyin (örneğin, cat yazarak /proc/meminfo;/proc dosya sistemi ve burada bulabileceğiniz bilgi türleri hakkında ayrıntılar için man proc yazın). İlk girişlerden biri /proc/meminfo, sisteminizdeki toplam bellek miktarıdır. Diyelim ki 8 GB gibi bir bellek; öyleyse, mem 4000'i (yaklaşık 4 GB) çalıştırarak ve takas giriş/çıkış sütunlarını izleyerek başlayın. Hiç sıfır olmayan değerler veriyorlar mı? Ardından, 5000, 6000, vb. ile deneyin. Program, ilk döngüye kıyasla ikinci döngüye (ve ötesine) girerken bu

değerlere ne olur? İkinci, üçüncü ve sonraki döngüler sırasında ne kadar veri (toplam) içeri ve dışarı değiştirilir? (rakamlar mantıklı mı?)

4. Yukarıdakiyle aynı deneyleri yapın, ancak şimdi diğer istatistikleri (CPU kullanımı ve G / Ç istatistiklerini engelleme gibi) izleyin. Mem çalışırken nasıl değişirler?

5. Şimdi performansı inceleyelim. Mem için belleğe rahatça uyan bir giriş seçin (sistemdeki bellek miktarı 8 GB ise 4000 diyelim). 0. döngü (ve sonraki 1., 2. döngüler vb.) ne kadar sürer? Şimdi bellek boyutunun ötesinde rahatça bir boyut seçin (8 GB bellek varsayarak tekrar 12000 diyelim). Döngüler burada ne kadar sürer? Bant genişliği sayıları nasıl karşılaştırılır? Her şeyi rahatça belleğe sığdırmakla sürekli yer değiştirirken performans arasındaki fark nedir? x ekseninde mem tarafından kullanılan belleğin boyutunu ve y ekseninde söz konusu belleğe erişmenin bant genişliğini gösteren bir grafik yapabilir misiniz? Son olarak, hem her şeyin belleğe sığdığı hem de sığmadığı durumda, ilk döngünün performansı sonraki döngülerin performansıyla nasıl karşılaştırılır?

6. Takas alanı sonsuz değildir. Ne kadar takas alanı olduğunu görmek için takas aracını –s bayrağıyla kullanabilirsiniz. Takasta mevcut görünenin ötesinde, mem'I giderek daha büyük değerlerle çalıştırmaya çalışırsanız ne olur? Bellek ayırma hangi noktada başarısız olur?

```
dave@howtogeek:~$ vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy
 5  0  12288  92448  81296 692400    0    3   351   312  128  544  3  2
dave@howtogeek:~$
```

programın bellek kullanımı öngörülenden daha fazla olabilir ve bellek ayırma ile ayrılan alan yetersiz kalabilir. Bu durumda, program bellekten daha fazla veri kullanmaya çalıştığında bellek aşımı hatası alır.

7. Son olarak, gelişmişseniz, sisteminizi swapon ve swapoff kullanarak farklı takas cihazları kullanacak şekilde yapılandırabilirsiniz. Ayrıntılar için kılavuz sayfalarını okuyunuz. Farklı donanımlara erişiminiz varsa, klasik bir sabit sürücüye, flash tabanlı bir SSD'ye ve hatta bir RAID dizisine geçiş yaparken takas performansının nasıl değiştiğini görün. Yeni cihazlarla takas performansı ne kadar arttırılabilir? Bellek içi performansa ne kadar yaklaşabilirsiniz?

Takas performansını etkileyen en önemli faktörlerden biri, takas alanı olarak kullanılan cihazın okuma/yazma hızıdır. Bu nedenle, sabit bir sürücüden flash tabanlı bir SSD'ye geçiş yapıldığında, takas performansı genellikle hızlanacaktır. Çünkü SSD'ler, sabit sürücülere göre daha hızlı okuma/yazma hızlarına sahiptirler. Ayrıca, bir RAID dizisi kullanılması da takas performansını artırabilir. Bu sistem, birden fazla diski paralel olarak kullanarak okuma/yazma işlemlerini hızlandırır ve bu sayede takas performansını artırabilir. Ancak, bu değişikliklerin ne kadar performans artışı sağlayacağı tam olarak tahmin edilemez ve her sistem için farklılık gösterebilir.

sabit bir sürücüden flash tabanlı bir SSD'ye geçiş yaparak veya bir RAID dizisi kullanarak bellek okuma/yazma hızını artırabilir ve bu sayede bellek içi performansı yakın bir seviyeye getirebilirsiniz. Ancak, bu yöntemler bellek içi performansı tamamen aynı seviyeye getiremez.