21

Fiziksel Merkezin Ötesinde: Mekanizmalar

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

Bunu yapmak için, (**memory hierarchy)** bellek hiyerarşisinde ek bir düzeye ihtiyacımız var. Buraya kadar tüm sayfaların fiziksel bellekte bulunduğunu varsaydık. Ancak, büyük adres alanlarını desteklemek için, işletim sisteminin şu anda çok fazla talep görmeyen adres alanlarının bölümlerini saklayacak bir yere ihtiyacı olacaktır. Genel olarak, böyle bir yerin özellikleri, hafızadan daha fazla kapasiteye sahip olması gerektiğidir; sonuç olarak, genellikle daha yavaştır (daha hızlı olsaydı, onu sadece bellek olarak kullanırdık, değil mi?). Modern sistemlerde bu role genellikle bir sabit disk sürücüsü hizmet eder. Bu nedenle, bellek hiyerarşimizde, büyük ve yavaş sabit diskler en altta, bellek hemen üstte yer alır. Ve böylece sorunun özüne geliyoruz:

SORUNUN MERKEZİ: FİZİKSEL HAFIZANIN ÖTESİNE NASIL GİDİLİR

İşletim sistemi, büyük bir sanal adres alanı yanılsamasını şeffaf bir şekilde sağlamak için daha büyük, daha yavaş bir cihazı nasıl kullanabilir?

Sorabileceğiniz bir soru: neden bir işlem için tek bir büyük adres alanını desteklemek istiyoruz? Bir kez daha, cevap rahatlık ve kullanım kolaylığıdır. İşletim sisteminin sağladığı ve hayatınızı büyük ölçüde kolaylaştırdığı güçlü bir yanılsamadır. Rica ederim! Programcıların kod parçalarını veya verileri gerektiği gibi belleğe girip çıkarmasını gerektiren (**memory overlays)** bellek bindirmeleri 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 sağlamanız gerekir; yuh!

YAN KONU: DEPOLAMA TEKNOLOJİLERİ

G/Ç cihazlarının gerçekte nasıl çalıştığını daha sonra çok daha derinlemesine inceleyeceğiz (G/Ç cihazlarıyla ilgili bölüme bakın). Bu yüzden sabırlı ol! Ve elbette daha yavaş olan cihazın bir sabit disk olması gerekmez, Flash tabanlı SSD gibi daha modern bir şey olabilir. Bunları da konuşacağız. Şimdilik fiziksel belleğin kendisinden bile daha büyük, çok büyük bir sanal bellek yanılsaması oluşturmamıza yardımcı olmak için kullanabileceğimiz büyük ve nispeten yavaş bir cihazımız olduğunu varsayalım.

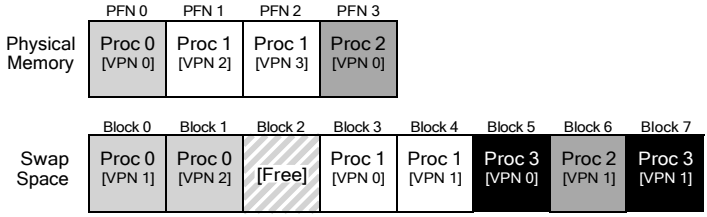
Tek bir işlemin ötesinde, takas alanının eklenmesi, işletim sisteminin aynı anda çalışan birden çok işlem için büyük bir sanal bellek yanılsamasını desteklemesine olanak tanır. Çoklu programlamanın icadı (makineyi daha iyi kullanmak için "bir kerede" birden fazla programı çalıştırmak) neredeyse bazı sayfaların değiş tokuş edilmesini gerektiriyordu, çünkü ilk makineler açıkça tüm işlemlerin ihtiyaç duyduğu tüm sayfaları bir kerede tutamadı. Bu nedenle, çoklu programlama ve kullanım kolaylığının birleşimi, fiziksel olarak mevcut olandan daha fazla bellek kullanımını desteklemek istememize yol açar. Bu, tüm modern VM sistemlerinin yaptığı bir şeydir; şimdi hakkında daha fazla öğreneceğimiz bir şey.

21.1 Takas Alanı

Yapmamız gereken ilk şey, sayfaları ileri geri taşımak için diskte biraz yer ayırmak. İşletim sistemlerinde, genellikle (**swap space)** takas alanı olarak adlandırılırız, çünkü bellekteki sayfaları onunla değiştiririz ve sayfaları buradan belleğe değiştiririz. Bu nedenle, işletim sisteminin sayfa boyutundaki birimlerde takas alanından okuyabildiğini ve yazabildiğini varsayacağız. Bunu yapmak için işletim sisteminin belirli bir sayfanın (**disk address)** disk adresini hatırlaması gerekir.

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

Minik örnekte (Şekil 21.1) 4 sayfalık fiziksel bellek ve 8 sayfalık takas alanının küçük bir örneğini görebilirsiniz. Örnekte, üç işlem (Proc 0, Proc 1 ve Proc 2) fiziksel belleği aktif olarak paylaşmaktadır; Bununla birlikte, üçünün her biri, yalnızca geçerli sayfalarından bazılarını bellekte bulundurur, geri kalanı ise diskteki takas alanında bulunur. Dördüncü bir işlem (Proc 3), tüm sayfalarını diske kaydırdı ve bu nedenle açıkça şu anda çalışmıyor. Bir blok takas serbest kalır. Bu küçük örnekten bile, umarım takas alanı kullanmanın sistemin belleğin gerçekte olduğundan daha büyük olduğunu iddia etmesine nasıl izin verdiğini görebilirsiniz. Takas alanının, trafiği takas etmek için disk üzerindeki tek konum olmadığını unutmamalıyız. Örneğin, bir program ikili dosyası çalıştırdığınızı varsayalım (örneğin, ls veya kendi derlenmiş ana programınız). Bu ikili dosyadaki kod sayfaları başlangıçta diskte bulunur ve program çalıştığında belleğe yüklenir (ya program yürütmeye



Şekil 21.1: **Fiziksel Hafıza ve Takas Alanı**

veya modern sistemlerde olduğu gibi, gerektiğinde bir sayfa). Ancak, sistemin diğer ihtiyaçlar için fiziksel bellekte yer açması gerekiyorsa, daha sonra bunları dosya sistemindeki disk üzerindeki ikili dosyadan tekrar değiştirebileceğini bilerek, bu kod sayfaları için bellek alanını güvenle yeniden kullanabilir.

21.2 Mevcut Bit

Artık diskte biraz yerimiz olduğuna göre, diske ve diskten sayfa değiş tokuşunu desteklemek için sistemde daha yüksek bir makine eklememiz gerekiyor. Basitlik için donanım tarafından yönetilen bir TLB'ye sahip bir sistemimiz olduğunu varsayalım.

Önce bir bellek referansında ne olduğunu hatırlayın. Çalışan süreç sanal bellek referansları oluşturur (talimat getirmeleri veya veri erişimleri için) ve bu durumda donanım, istenen verileri bellekten almadan önce bunları fiziksel adreslere çevirir. Donanımın önce VPN'yi sanal adresten çıkardığını, TLB'de eşleşme (a **TLB hit**) (TLB isabeti) olup olmadığını kontrol ettiğini ve bir isabet durumunda ortaya çıkan fiziksel adresi üretip bellekten aldığını unutmayın. Hızlı olduğu için (ek bellek erişimi gerektirmeyen) bu durum umarım genel durumdur. TLB'de VPN bulunamazsa (i.e., a **TLB miss**) (yani bir TLB eksik), donanım bellekteki sayfa tablosunu bulur (using the **page table base register**) (sayfa tablosu taban kaydını kullanarak) ve VPN'i aşağıdaki gibi kullanarak bu sayfa için **page table entry (PTE)** sayfa tablosu girişini indeks (PTE) arar. Sayfa geçerliyse ve fiziksel bellekte mevcutsa, donanım PFN'yi PTE'den çıkarır, TLB'ye yükler ve yeniden dener. Talimat, bu sefer bir TLB isabeti oluşturuyor; Şimdiye kadar, çok iyi.

YAN KONU: TAKAS TERMİNOLOJİSİ VE DİĞER ŞEYLER

Sanal bellek sistemlerindeki terminoloji, makineler ve işletim sistemleri arasında biraz kafa karıştırıcı ve değişken olabilir. Örneğin, daha genel olarak bir sayfa hatası (**page fault)** bir tür hata oluşturan bir sayfa tablosuna yapılan herhangi bir referansa atıfta bulunabilir: bu, burada tartıştığımız hata türünü, yani sayfa olmayan bir hatayı içerebilir, ancak bazen yasadışı bellek erişimlerine bakın. Gerçekten de, kesinlikle yasal erişim (bir işlemin sanal adres alanına eşlenmiş, ancak o sırada fiziksel bellekte olmayan bir sayfaya) kesinlikle "hata" olarak adlandırmamız tuhaftır; gerçekten, buna sayfa kaçırma (**page miss**) denmeli. Ancak çoğu zaman, insanlar bir programın "sayfa hatası" olduğunu söylediğinde, işletim sisteminin diske değiştirdiği sanal adres alanının bölümlerine eriştiği anlamına gelir.

Bu davranışın "arıza" olarak bilinmesinin nedeninin, işletim sistemindeki makinenin onu ele almasıyla ilgili olduğundan şüpheleniyoruz. Olağandışı bir şey olduğunda, yani donanımın nasıl ele alacağını bilmediği bir şey meydana geldiğinde, donanım, işleri daha iyi hale getirebileceğini umarak kontrolü işletim sistemine aktarır. Bu durumda, bir işlemin erişmek istediği sayfa bellekte eksiktir; donanım, bir istisna oluşturan tek şeyi yapar ve işletim sistemi oradan devralır. Bu, bir süreç yasa dışı bir şey yaptığında olanla aynı olduğundan, faaliyeti “hata” olarak adlandırmamız belki de şaşırtıcı değildir.

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

21.3 Sayfa Hatası

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). Her iki sistem türünde de, bir sayfa yoksa, sayfa hatasını işlemek için işletim sistemi sorumludur. Uygun şekilde adlandırılmış işletim sistemi sayfa hatası işleyicisi **(page-fault handler)** ne yapılacağını belirlemek için çalışır. Hemen hemen tüm sistemler yazılımdaki sayfa hatalarını işler; donanım tarafından yönetilen bir TLB ile bile donanım, bu önemli görevi yönetmek için işletim sistemine güvenir. Bir sayfa yoksa ve diske değiştirilmişse, işletim sisteminin sayfa hatasına hizmet vermek için sayfayı belleğe alması 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.

YAN KONU: DONANIM SAYFA HATALARINI NEDEN İŞLEMİYOR

TLB ile olan deneyimimizden, donanım tasarımcılarının her şeyi yapmak için işletim sistemine güvenmekten nefret ettiğini biliyoruz. Öyleyse neden bir sayfa hatasını işlemek için işletim sistemine güveniyorlar? Birkaç ana sebep var. İlk olarak, diskteki sayfa hataları yavaştır; İşletim sisteminin tonlarca talimatı yürüterek bir hatayı işlemesi 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 I/O'ların nasıl verileceğini ve şu anda hakkında fazla bir şey bilmediği birçok başka ayrıntıyı anlaması gerekir. Bu nedenle, hem performans hem de basitlik nedeniyle işletim sistemi sayfa hatalarını yönetir ve hatta donanım türleri bile mutlu olabilir.

Disk G/Ç tamamlandığında, işletim sistemi daha sonra sayfayı mevcut olarak işaretlemek için sayfa tablosunu günceller, yeni alınan sayfanın bellek içi konumunu kaydetmek için sayfa tablosu girişinin (PTE) PFN alanını günceller, ve talimatı yeniden deneyin. Bu sonraki girişim, daha sonra hizmet verilecek ve çeviriyle TLB'yi güncelleyecek bir TLB ıskası oluşturabilir (bu adımdan kaçınmak için sayfa hatasına hizmet verirken alternatif olarak TLB güncellenebilir). Son olarak, son bir yeniden başlatma TLB'deki çeviriyi bulur ve böylece istenen veriyi veya talimatı çevrilmiş fiziksel adresteki bellekten almaya devam eder.

G/Ç uçuş halindeyken işlemin bloke **(blocked)** olacağını unutmayın. durum. Böylece işletim sistemi, sayfa hatasına hizmet verilirken diğer hazır işlemleri çalıştırmakta serbest olacaktır. G/Ç pahalı olduğu için, bir işlemin G/Ç'sinin (sayfa hatası) bu örtüşmesi**(overlap)** ve diğerinin yürütülmesi, çok programlı bir sistemin donanımını en verimli şekilde kullanmasının başka bir yoludur.

21.4 Ya Bellek Doluysa

Yukarıda açıklanan süreçte, takas alanından bir sayfada (**page in)** sayfalamak için bol miktarda boş bellek olduğunu varsaydığımızı fark edebilirsiniz. Tabii ki, durum böyle olmayabilir; bellek dolu (veya buna yakın) olabilir. Bu nedenle, işletim sistemi, işletim sisteminin getireceği yeni sayfa(lar)a yer açmak için bir veya daha fazla sayfayı ilk sayfadan çıkarmak isteyebilir (**page out)** . Çıkarılacak veya değiştirilecek (**replace)** bir sayfa seçme işlemi, sayfa değiştirme politikası (**page-replacement policy)** olarak bilinir.

Anlaşıldığı üzere, iyi bir sayfa değiştirme politikası oluşturmak için çok düşünülmüş, çünkü yanlış sayfayı atmak program performansı üzerinde büyük bir maliyete neden olabilir. 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 bu. Şimdilik, burada açıklanan mekanizmaların üzerine inşa edilmiş böyle bir politikanın var olduğunu anlamak yeterlidir.

.

```
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)
```

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

21.5 Sayfa Hatası Kontrol Akışı

Tüm bu bilgiler yerindeyken, 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 fikre sahip olmalısınız. Daha fazla ayrıntı için Şekil 21.2 ve 21.3'teki kontrol akışına bakın; ilk şekil, donanımın çeviri sırasında ne yaptığını ve ikincisi, işletim sisteminin bir sayfa hatası üzerine ne yaptığını gösterir.

Şekil 21.2'deki donanım kontrol akış şemasından, artık bir TLB ıskasının oluştuğunu anlamak için üç önemli durum olduğuna dikkat edin. Birincisi, sayfanın hem mevcut (**present**) hem de geçerli (**valid**) olması (Satır 18-21); bu durumda, TLB ıska işleyici, PFN'yi PTE'den 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 erişilmesi için meşru bir sayfa olmasına rağmen (sonuçta geçerlidir), fiziksel bellekte mevcut değildir. Üçüncüsü (ve son olarak), örneğin programdaki bir hata nedeniyle (Satır 13-14) erişim geçersiz bir sayfaya olabilir. Bu durumda, PTE'deki başka hiçbir bit gerçekten önemli değildir; donanım bu geçersiz erişimi yakalar ve işletim sistemi tuzak işleyicisi çalışır ve büyük olasılıkla soruna neden olan işlemi sonlandırır.

Şekil 21.3'teki yazılım kontrol akışından, işletim sisteminin sayfa hatasına hizmet etmek için kabaca ne yapması gerektiğini görebiliriz. İlk olarak, işletim sistemi, yakında arızalanacak sayfanın içinde yer alması için fiziksel bir çerçeve bulmalıdır; eğer böyle bir sayfa yoksa, değiştirme algoritmasının çalışmasını ve bazı sayfaları bellekten atmasını beklememiz ve böylece onları burada kullanım için serbest bırakmamız gerekecek.

```
1  PFN = FindFreePhysicalPage()
2  if (PFN == -1)              // no free page found
3      PFN = EvictPage()       // run replacement algorithm
4  DiskRead(PTE.DiskAddr, PFN) // sleep (waiting for I/O)
5  PTE.present = True          // update page table with present
6  PTE.PFN     = PFN           // bit and translation (PFN)
7  RetryInstruction()          // retry instruction
```

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

Elinde fiziksel bir çerçeve varken, işleyici daha sonra takas alanından sayfada okumak için G/Ç isteği gönderir. Son olarak, bu yavaş işlem tamamlandığında, işletim sistemi sayfa tablosunu günceller ve talimatı yeniden dener. Yeniden deneme, bir TLB ıskasıyla ve ardından başka bir yeniden denemede, donanımın istenen öğeye erişebileceği bir TLB isabetiyle sonuçlanacaktır.

21.6 Değiştirmeler Gerçekten Meydana Geldiğinde

Şimdiye kadar, değiştirmelerin nasıl gerçekleştiğini açıkladığımız yol, işletim sisteminin bellek tamamen dolana kadar beklediğini ve ancak o zaman başka bir sayfaya yer açmak için bir sayfayı değiştirdiğini (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 sisteminde, sayfaları bellekten ne zaman çıkarmaya başlayacağına karar vermeye yardımcı olmak için bir tür yüksek filigran (**high watermark)**(HW) ve düşük filigran **(low watermark)** (LW) bulunur. Bunun nasıl çalıştığı şu şekildedir: İşletim sistemi, LW'den daha az sayfanın bulunduğunu fark ettiğinde, belleği boşaltmaktan sorumlu bir arka plan iş parçacığı çalışır. Kullanılabilir HW sayfaları olana kadar iş parçacığı sayfaları çıkarır. Bazen takas arka plan programı **(swap daemon)** veya sayfa arka plan programı[1] **(page daemon)**[1] olarak adlandırılan arka plan iş parçacığı, daha sonra çalışan işlemler ve kullanılacak işletim sistemi için bir miktar bellek boşalttığı için mutlu bir şekilde uyku moduna geçer.

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

Arka plan sayfalama iş parçacığı ile çalışmak için, Şekildeki kontrol akışı 21.3 biraz değiştirilmelidir; Algoritma, doğrudan değiştirme yapmak yerine, boş sayfa olup olmadığını kontrol eder. Değilse, arka planda çağrı dizisine boş sayfaların gerekli olduğunu bildirir; iş parçacığı bazı sayfaları serbest bıraktığında, orijinal iş parçacığını yeniden uyandırır, bu da daha sonra istenen sayfaya girip işine devam edebilir.

[1]Genellikle "iblis" olarak telaffuz edilen "arka plan programı" sözcüğü, yararlı bir şey yapan bir arka plan dizisi veya işlemi için eski bir terimdir. (Bir kez daha!) Terimin kaynağının Multics [CS94] olduğu ortaya çıktı.

İPUCU: ARKA PLANDA ÇALIŞIN

Yapacak bir işiniz olduğunda, verimliliği artırmak ve işlemlerin gruplandırılmasına olanak sağlamak için bunu arka planda yapmak genellikle iyi bir fikirdir. İşletim sistemleri genellikle arka planda **(background)** çalışır; örneğin, birçok sistem arabellek dosyası, verileri diske yazmadan önce belleğe yazar. Bunu yapmanın pek çok olası faydası vardır: artan disk verimliliği, çünkü disk artık birçok yazıyı aynı anda alabilir ve böylece bunları daha iyi zamanlayabilir; uygulama, yazma işlemlerinin oldukça hızlı tamamlandığını düşündüğünden, yazma gecikmesi artırıldı; yazma işlemlerinin asla diske gitmesi gerekmeyebileceğinden (yani, dosya silinirse); ve arka plan çalışması muhtemelen sistem boştayken yapılabileceğinden **(idle time)** boş zamanın daha iyi kullanılması, böylece donanımın daha iyi kullanılması [G+95].

21.7 Özet

Bu kısa bölümde, bir sistemde fiziksel olarak mevcut olandan daha fazla belleğe erişme kavramını tanıttık. Bunu yapmak, sayfa tablosu yapılarında daha fazla karmaşıklık gerektirir, çünkü bize sayfanın bellekte mevcut olup olmadığını söylemek için mevcut bir bitin **(present bit)** (ya da benzer bir türün) 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!), tüm bu eylemlerin şeffaf **(transparently)** bir şekilde gerçekleştiğini hatırlayın. Süreç söz konusu olduğunda, sadece kendine özel, bitişik sanal belleğine erişiyor. Perde arkası, sayfalar fiziksel bellekte rastgele (bitişik olmayan) konumlara yerleştirilir ve bazen bellekte bile bulunmazlar, bu da diskten bir getirme gerektirir. Yaygın olarak bir hafıza erişiminin hızlı olmasını umarız. Bazı durumlarda hizmet vermek için birden çok disk işlemi gerekir; tek bir talimatı yerine getirmek kadar basit bir şey, en kötü durumda, tamamlanması birkaç milisaniye sürer.

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 (Ölçüm)

Bu ev ödevi size yeni bir araç olan vmstat'ı (**vmstat)** ve bunun bellek, CPU 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, bir pencerede ve diğerinde kolayca bir şeyler çalıştırabilmeniz için aynı makineye iki ayrı terminal bağlantısı açın.

   Şimdi, bir pencerede, makine kullanımıyla ilgili istatistikleri her saniye gösteren vmstat 1'i çalıştırın. Çıktısını anlayabilmek için man 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 rakamlar mantıklı mı? Aynı anda birden fazla mem örneği çalıştırıldığında bu nasıl değişir?

vmstat 1'i çalıştırdığımızda oluşan ekran görüntüsü aşağıdaki gibidir.

```
cagatay@ubuntu:~/Desktop/ostep/ostep-homework/vm-beyondphys$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 2085392  52476 977820    0    0    85    36   82  173  5  1 94  0  0
 0  0      0 2085368  52476 977820    0    0     0     0   77  235  1  0 100  0  0
 0  0      0 2085368  52476 977820    0    0     0     0   67  234  1  0 99  0  0
 0  0      0 2085244  52476 977820    0    0     0     0   56  145  1  0 100  0  0
 0  0      0 2085244  52476 977820    0    0     0     0   61  129  0  0 100  0  0
 0  0      0 2085244  52476 977820    0    0     0     0   71  189  1  0 99  0  0
 0  0      0 2085244  52476 977820    0    0     0     0   59  152  0  1 100  0  0
 0  0      0 2085120  52476 977820    0    0     0     0   54  135  1  0 99  0  0
 0  0      0 2085244  52476 977820    0    0     0     0  121  193  0  0 100  0  0
 0  0      0 2085244  52476 977820    0    0     0     0   99  162  1  0 100  0  0
 0  0      0 2085244  52476 977820    0    0     0     0   87  148  1  1 99  0  0
 0  0      0 2085244  52476 977820    0    0     0     0  118  186  0  0 100  0  0
 0  0      0 2085244  52476 977820    0    0     0     0   88  160  0  0 99  0  0
 0  0      0 2085252  52476 977812    0    0     0     0  107  183  1  0 99  0  0
 0  0      0 2085252  52476 977812    0    0     0     0   93  148  1  0 100  0  0
 0  0      0 2085252  52476 977812    0    0     0     0  109  183  0  1 100  0  0
 0  0      0 2085252  52476 977812    0    0     0     0  120  190  1  0 99  0  0
 0  0      0 2085252  52476 977812    0    0     0     0   85  161  1  0 100  0  0
 0  0      0 2085252  52476 977812    0    0     0     0   84  230  1  0 99  0  0
 0  0      0 2085252  52476 977812    0    0     0     0  310  496  1  1 99  0  0
 0  0      0 2085252  52476 977812    0    0     0     0  167  271  1  0 99  0  0
 0  0      0 2085252  52476 977812    0    0     0     0  175  294  1  0 99  0  0
 0  0      0 2085324  52476 977740    0    0     0     0   73  189  0  0 100  0  0
 0  0      0 2085324  52476 977740    0    0     0     0   65  149  1  0 99  0  0
 0  0      0 2085324  52476 977740    0    0     0     0  143  386  0  1 99  0  0
```

./mem 1'i çalıştırdığımızda oluşan ekran görüntüsü aşağıdaki gibidir.

```
cagatay@ubuntu:~/Desktop/ostep/ostep-homework/vm-beyondphys$ ./mem 1
allocating 1048576 bytes (1.00 MB)
  number of integers in array: 262144
loop 0 in 1.18 ms (bandwidth: 843.92 MB/s)
loop 403 in 0.48 ms (bandwidth: 2074.33 MB/s)
loop 807 in 0.49 ms (bandwidth: 2049.00 MB/s)
loop 1212 in 0.49 ms (bandwidth: 2061.08 MB/s)
loop 1609 in 0.49 ms (bandwidth: 2054.02 MB/s)
loop 2015 in 0.48 ms (bandwidth: 2071.26 MB/s)
loop 2419 in 0.48 ms (bandwidth: 2065.14 MB/s)
loop 2814 in 0.49 ms (bandwidth: 2024.28 MB/s)
loop 3216 in 0.49 ms (bandwidth: 2028.19 MB/s)
loop 3615 in 0.54 ms (bandwidth: 1865.79 MB/s)
loop 4016 in 0.48 ms (bandwidth: 2074.33 MB/s)
loop 4422 in 0.48 ms (bandwidth: 2062.10 MB/s)
loop 4828 in 0.55 ms (bandwidth: 1814.93 MB/s)
loop 5231 in 0.47 ms (bandwidth: 2105.57 MB/s)
loop 5634 in 0.48 ms (bandwidth: 2070.24 MB/s)
loop 6033 in 0.49 ms (bandwidth: 2045.00 MB/s)
loop 6428 in 0.49 ms (bandwidth: 2029.17 MB/s)
loop 6832 in 0.48 ms (bandwidth: 2074.33 MB/s)
loop 7232 in 0.49 ms (bandwidth: 2033.11 MB/s)
loop 7631 in 0.49 ms (bandwidth: 2054.02 MB/s)
loop 8037 in 0.48 ms (bandwidth: 2104.52 MB/s)
loop 8440 in 0.48 ms (bandwidth: 2062.10 MB/s)
loop 8840 in 0.49 ms (bandwidth: 2024.28 MB/s)
loop 9239 in 0.49 ms (bandwidth: 2024.28 MB/s)
```

İşlem çalışırken kullanıcı zamanlaması artar. Daha fazla örnek, kullanıcı zamanlamasını doğrusal olarak artırır. Ancak daha yavaş tek çekirdekli bir makinede değil. Kullanıcı zamanlaması yalnızca bir örnekle 100'e ulaşır. Bu mantıklı.

Bağlam değiştirme ve kesintilerin sayısı da önemli ölçüde artar. Bu biraz tuhaf. TLB kaçırmaları, kesintilere veya bağlam değişikliğine neden olmamalıdır. Programı çalıştırmadan önce kullanıcı zamanlaması ve sistem zamanlaması neredeyse sıfırdı. Yani diğer programların çalışmasına izin verecekse artış bu kadar olmamalı.

2. Şimdi mem'i çalıştırırken bazı hafıza istatistiklerine bakmaya başlayalım. İki sütuna odaklanacağız: swpd (kullanılan sanal bellek miktarı) ve ücretsiz (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 konusunda nelere dikkat ediyorsunuz? Özellikle, programdan çıkıldığında serbest sütunu nasıl değişir? Mem çıkışında boş hafıza miktarı beklenen miktarda artıyor mu?

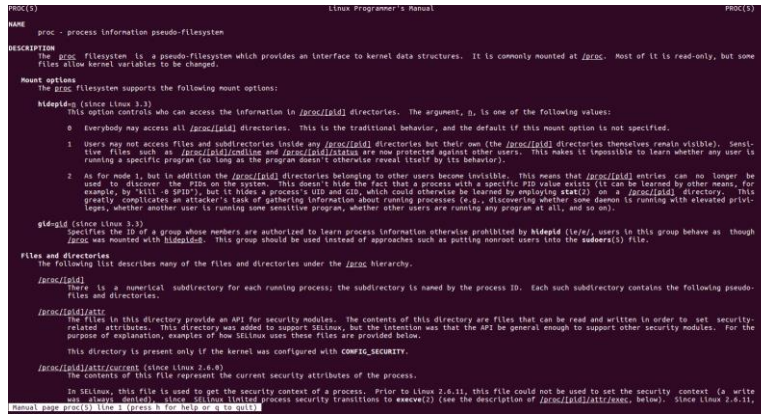
./mem 1024'ü çalıştırdığımızda oluşan ekran görüntüsü aşağıdaki gibidir.

```
cagatay@ubuntu:~/Desktop/ostep/ostep-homework/vm-beyondphys$ ./mem 1024
allocating 1073741824 bytes (1024.00 MB)
  number of integers in array: 268435456
loop 0 in 1235.19 ms (bandwidth: 829.02 MB/s)
loop 1 in 544.67 ms (bandwidth: 1880.03 MB/s)
loop 2 in 540.34 ms (bandwidth: 1895.12 MB/s)
loop 3 in 535.44 ms (bandwidth: 1912.46 MB/s)
loop 4 in 522.00 ms (bandwidth: 1961.69 MB/s)
loop 5 in 545.24 ms (bandwidth: 1878.07 MB/s)
loop 6 in 548.01 ms (bandwidth: 1868.60 MB/s)
loop 7 in 572.02 ms (bandwidth: 1790.13 MB/s)
loop 8 in 579.38 ms (bandwidth: 1767.42 MB/s)
loop 9 in 567.92 ms (bandwidth: 1803.06 MB/s)
loop 10 in 569.20 ms (bandwidth: 1799.02 MB/s)
loop 11 in 559.71 ms (bandwidth: 1829.54 MB/s)
loop 12 in 551.22 ms (bandwidth: 1857.70 MB/s)
loop 13 in 567.81 ms (bandwidth: 1803.41 MB/s)
loop 14 in 557.29 ms (bandwidth: 1837.47 MB/s)
loop 15 in 561.60 ms (bandwidth: 1823.36 MB/s)
loop 16 in 547.46 ms (bandwidth: 1870.45 MB/s)
loop 17 in 559.80 ms (bandwidth: 1829.23 MB/s)
loop 18 in 543.32 ms (bandwidth: 1884.72 MB/s)
loop 19 in 553.98 ms (bandwidth: 1848.46 MB/s)
loop 20 in 550.44 ms (bandwidth: 1860.34 MB/s)
```

Serbest miktar tam olarak beklendiği gibi artmıyor. Serbest miktarın eski haline dönmesi bekleniyordu. 1GB RAM'e sahip bir sistemde ./mem 512 ile çalıştırıldığında artışta birkaç yüz kilobaytlık eksiklik vardı. Kullanılan bellek RAM'e sığdığından takas değişmez.

3. Daha sonra, diske ve diskten ne kadar değiş tokuş yapıldığını gösteren takas sütunlarına bakacağız. Tabii ki, bunları etkinleştirmek için mem'i büyük miktarda bellekle çalıştırmanız gerekir. İlk olarak, Linux sisteminizde ne kadar boş bellek olduğunu inceleyin (örneğin, cat yazarak) /proc/meminfo; /proc dosya sistemi ve orada 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 ikinci döngüye (ve ötesine) girdiğinde, ilk döngüye kıyasla bu değerlere ne olur? İkinci, üçüncü ve sonraki döngüler sırasında ne kadar veri (toplam) içeri ve dışarı değiştirilir? (sayılar mantıklı mı?)

man proc yazdığımızda oluşan ekran görüntüsü aşağıdak gibidir. Dosya sistemi ve oradaki bilgi türleri ile ilgili ayrıntılar yer alıyor.



./mem 4000'i çalıştırdığımızda oluşan ekran görüntüsü aşağıdaki gibidir.

```
cagatay@ubuntu:~/Desktop/ostep/ostep-homework/vm-beyondphys$ ./mem 4000
allocating 4194304000 bytes (4000.00 MB)
  number of integers in array: 1048576000
loop 0 in 14300.91 ms (bandwidth: 279.70 MB/s)
loop 1 in 24772.23 ms (bandwidth: 161.47 MB/s)
loop 2 in 24000.37 ms (bandwidth: 166.66 MB/s)
loop 3 in 23674.98 ms (bandwidth: 168.95 MB/s)
loop 4 in 24547.43 ms (bandwidth: 162.95 MB/s)
loop 5 in 24037.55 ms (bandwidth: 166.41 MB/s)
```

Örnekte istenilen gibi daha yüksek değer girdiğimde memory allocation failed hatası verdi.

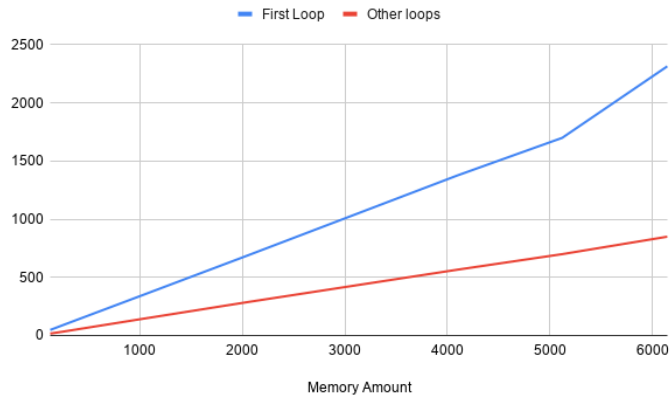
```
cagatay@ubuntu:~/Desktop/ostep/ostep-homework/vm-beyondphys$ ./mem 8000
allocating 8388608000 bytes (8000.00 MB)
memory allocation failed
cagatay@ubuntu:~/Desktop/ostep/ostep-homework/vm-beyondphys$
```

İlk döngü sırasında büyük bir miktar takas edilir **(swap out)** ve küçük bir miktar takas edilir **(swap in)**. Ve sonra takas değeri düşer ve sonunda sıfır olur ve sıfır olarak kalır. Bu azalma lineer değildir. Bazen düşer ve tekrar artar. Ama sonunda 0'a kadar yerleşir. Vurgulu takas yerleşmez. Böylece sistem, büyük paya yer açmak için önce büyük bir miktarı takas eder. Ardından, minimum sayıda boş alanı (HW) korumak için daha fazlasını değiştirmeye devam eder. Bu hedefe ulaşıldığında artık hiçbir şeyi takas etme. Öte yandan, sistem her zaman bazı küçük miktarları arada sırada diğer işlemler için takas etmek zorundadır. Takas etmeyi bıraktıktan sonra, küçük miktarları takas etmek LW'ye dokunmaz. Böylece sistem başka bir şeyi takas etmeye çalışmaz.

4. Yukarıdaki deneylerin aynısını yapın, ancak şimdi diğer istatistikleri (CPU kullanımı ve blok G/Ç istatistikleri gibi) izleyin. Mem çalışırken nasıl değişirler?

   İlk başta, takasın olmadığı zamana kıyasla biraz daha düşük. Ama sy biraz daha yüksek. Sonra wa gözlemlenir. Sistem, değiştirmeden önce hangi sayfaların değiştirileceğine karar vererek bazı işlemler yapmış olmalıdır. Bittiğinde, sayfalar diske yazılırken bir miktar bekleme gerçekleşir.

5. Şimdi performansı inceleyelim. Bellek için belleğe rahatça uyan bir giriş seçin (sistemdeki bellek miktarı 8 GB ise 4000 deyin). 0. döngü (ve sonraki 1., 2. döngüler vb.) ne kadar sürer? Şimdi bellek boyutunun ötesinde rahat bir boyut seçin (yine 8 GB bellek varsayarak 12000 deyin hafıza). Buradaki döngüler ne kadar sürüyor? 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. Mem'i, takasta mevcut görünenin ötesinde, giderek daha büyük değerlerle çalıştırmayı denerseniz ne olur? Bellek tahsisi hangi noktada başarısız oluyor?

   Bellekte ve takasta mevcut olandan daha fazlasını ayırmaya çalışmak, süreci öldürüyor gibi görünüyor.

7. Son olarak, ileri düzeydeyseniz, takas ve takas kullanarak sisteminizi farklı takas cihazları kullanacak şekilde yapılandırabilirsiniz. Ayrıntılar için kılavuz sayfalarını okuyun. Farklı donanımlara erişiminiz varsa, klasik bir sabit sürücüye, flash tabanlı bir SSD'ye ve hatta bir RAID dizisine geçtiğinizde değiştirme performansının nasıl değiştiğini görün. Daha yeni cihazlar aracılığıyla takas performansı ne kadar iyileştirilebilir? Bellek içi performansa ne kadar yaklaşabilirsiniz?

   Bir HDD'den 2 GB'lık bir takas kullanıldığında, bir SSD'ye kıyasla ilk döngülerin tamamlanması 10 kat daha fazla zaman alır.