CH 21

Fiziksel Hafızanın Ötesinde: Mekanizmalar

Şimdiye kadar, bir adres alanının gerçekçi olmayan bir şekilde küçük olduğunu ve fiziksel belleğe sığdığını varsaydık. Aslında, çalışan her sürecin her adres alanının belleğe sığdığını varsayıyorduk. Şimdi bu büyük varsayımları gevşeteceğiz ve eşzamanlı olarak çalışan birçok büyük adres alanını desteklemek istediğimizi varsayacağız.
Bunu yapmak için **bellek hiyerarşisinde** ek bir seviyeye ihtiyacımız var. Şimdiye kadar, tüm sayfaların fiziksel bellekte bulunduğunu varsaydık. Ancak, büyük adres alanlarını desteklemek için, işletim sisteminin adres alanlarının şu anda çok fazla kullanılmayan kısımlarını saklamak için bir yere ihtiyacı olacaktır. Genel olarak, böyle bir yerin özellikleri bellekten daha fazla kapasiteye sahip olması gerektiğidir; sonuç olarak, genellikle daha yavaştır (daha hızlı olsaydı, onu sadece bellek olarak kullanırdık, değil mi?). Modern sistemlerde bu rol genellikle bir **sabit disk sürücüsü** tarafından yerine getirilir. Böylece, bellek hiyerarşimizde, büyük ve yavaş sabit diskler en altta, bellek ise

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

hemen üstünde yer alır. Ve böylece sorunun özüne ulaşmış oluyoruz:

Aklınıza gelebilecek bir soru: neden bir süreç için tek bir büyük adres alanını desteklemek istiyoruz? Cevap bir kez daha kolaylık ve kullanım kolaylığıdır. Geniş bir adres alanıyla, programınızın veri yapı için bellekte yeterince yer olup olmadığı konusunda endişelenmenize gerek kalmaz; bunun yerine, programı doğal bir şekilde yazarsınız ve gerektiğinde bellek ayırırsınız. Bu, işletim sisteminin sağladığı güçlü bir yanılsamadır ve hayatınızı büyük ölçüde kolaylaştırır. Bir şey değil! Buna karşılık, programcıların kod veya veri parçalarını ihtiyaç duyuldukça belleğe elle sokup çıkarmalarını gerektiren **bellek** kaplamalarını kullanan eski sistemlerde buna benzer bir durum söz konusudur [D97]. Bunun nasıl bir şey olduğunu hayal etmeye çalışın: bir işlevi çağırmadan ya da bazı verilere erişmeden önce, kodun ya da verinin bellekte olmasını sağlamanız gerekir; iğrenç!

BIR KENARA: DEPOLAMA TEKNOLOJILERI

I/O aygıtlarının gerçekte nasıl çalıştığını daha sonra çok daha derinlemesine inceleyeceğiz (I/O aygıtları bölümüne bakın). Bu yüzden sabırlı olun! Ve elbette daha yavaş aygıtın bir sabit disk olması gerekmez, ancak Flash tabanlı bir SSD gibi daha modern bir şey olabilir. Bunlar hakkında da konuşacağız. Şimdilik, fiziksel bellekten bile daha büyük, çok büyük bir sanal bellek yanılsaması oluşturmamıza yardımcı olması için kullanabileceğimiz büyük ve nispeten yavaş bir aygıtımız olduğunu varsayın.

Tek bir işlemin ötesinde, takas alanının eklenmesi işletim sisteminin aynı anda çalışan birden fazla işlem için büyük bir sanal bellek yanılsamasını desteklemesini sağlar. Çoklu programlamanın icadı (makineyi daha iyi kullanmak için "aynı anda" birden fazla programın çalıştırılması) neredeyse bazı sayfaların takas edilebilmesini gerektiriyordu, çünkü ilk makineler açıkça tüm işlemlerin ihtiyaç duyduğu tüm sayfaları aynı anda tutamıyordu. Bu nedenle, çoklu programlama ve kullanım kolaylığı kombinasyonu, fiziksel olarak mevcut olandan daha fazla bellek kullanımını desteklemek istememize yol açmaktadır. Bu, tüm modern sanal makine sistemlerinin yaptığı bir şeydir; şimdi bu konuda daha fazla şey öğreneceğiz.

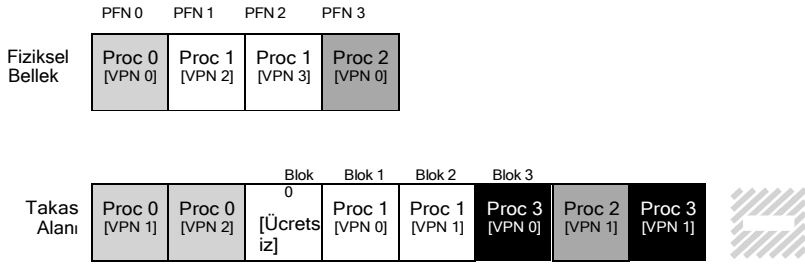
Takas Alanı

Yapmamız gereken ilk şey, sayfaları ileri geri taşımak için diskte bir miktar alan ayırmaktır. İşletim sistemlerinde bu alana genellikle **takas alanı diyoruz,** çünkü sayfaları bellekten bu alana *takas ediyoruz* ve sayfaları bu alandan belleğe *takas ediyoruz.* Bu nedenle, işletim sisteminin sayfa boyutlu birimler halinde takas alanından okuyabildiğini ve bu alana yazabildiğini varsayacağız. Bunu yapmak için işletim sisteminin belirli bir sayfanın **disk** adresini hatırlaması gerekecektir.

Takas alanının boyutu önemlidir, çünkü sonuçta belirli bir zamanda bir sistem tarafından kullanılabilecek maksimum bellek sayfası sayısını belirler. Basit olması 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 süreç (Proc 0, Proc 1 ve Proc 2) aktif olarak fiziksel belleği paylaşmaktadır; ancak her üçünün de geçerli sayfalarının yalnızca bir kısmı bellekte, geri kalanı ise diskteki takas alanında bulunmaktadır. Dördüncü bir süreç (Proc 3) tüm sayfalarını diske takas etmiştir ve bu nedenle şu anda çalışmadığı açıktır. Bir takas bloğu boş kalmıştır. Bu küçük örnekten bile, takas alanı kullanmanın sistemin belleği gerçekte olduğundan daha büyükmüş gibi göstermesine nasıl izin verdiğini görebileceğinizi umuyoruz.

Takas alanının takas trafiği için disk üzerindeki tek konum olmadığını belirtmeliyiz. Örneğin, bir program ikilisi (örneğin, `ls` veya kendi derlenmiş `ana programınız`) çalıştırdığınızı varsayalım. Bu ikilinin kod sayfaları başlangıçta diskte bulunur ve program çalıştığında belleğe yüklenir (ya program yürütülmeye başladığında hepsi birden,



Fiziksel Bellek ve Takas Alanı

ya da modern sistemlerde olduğu gibi, gerektiğinde her seferinde bir sayfa). Ancak, sistemin diğer ihtiyaçları için fiziksel bellekte yer açması gerekiyorsa, bu kod sayfaları için bellek alanını güvenli bir şekilde yeniden kullanabilir ve daha sonra bunları dosya sistemindeki disk içi ikili dosyadan tekrar takas edebileceğini bilir.

Şimdiki Zaman

Artık diskte biraz yerimiz olduğuna göre, sayfaların diske takas edilmesini desteklemek için sistemde daha yukarılara bazı ma- kineler eklememiz gerekir. Basit olması için donanım tarafından yönetilen TLB'ye sahip bir sistemimiz olduğunu varsayalım.

Öncelikle bir bellek referansında ne olduğunu hatırlayın. Çalışma süreci sanal bellek referansları üretir (komut getirme veya veri erişimleri için) ve bu durumda donanım bellekten istenen veriyi almadan önce bunları fiziksel adreslere çevirir.

Donanımın önce VPN'i sanal adresten çıkardığını, TLB'de bir eşleşme olup olmadığını kontrol ettiğini (bir **TLB isabeti**) ve bir isabet varsa, ortaya çıkan fiziksel adresi ürettiğini ve bellekten getirdiğini unutmayın. Bu, hızlı olduğu için (ek bellek erişimi gerektirmez) umarım yaygın bir durumdur. VPN TLB'de bulunamazsa (yani **TLB ıskalanırsa**), donanım bellekteki sayfa tablosunu bulur (sayfa tablosu **temel** kaydını kullanarak) ve VPN'i bir dizin olarak kullanarak bu sayfa için **sayfa tablosu girişini (PTE)** arar. Sayfa geçerliyse ve fiziksel bellekte mevcutsa, donanım PTE'den PFN'yi çıkarır, TLB'ye yükler ve yeniden dener
komutu, bu kez bir TLB vuruşu oluşturuyor; buraya kadar her şey yolunda.

Ancak sayfaların diske takas edilmesine izin vermek istiyorsak, daha da fazla makine eklememiz gerekir. Özellikle, donanım PTE'ye baktığında, sayfanın fiziksel bellekte *bulunmadığını görebilir. Donanımın* (ya da yazılım tarafından yönetilen bir TLB yaklaşımında işletim sisteminin) bunu belirleme yolu, her sayfa tablosu girişinde **mevcut bit olarak** bilinen yeni bir bilgi parçasıdır. Mevcut biti bire ayarlanmışsa, sayfa fiziksel bellekte mevcut demektir ve her şey yukarıdaki gibi devam eder; sıfıra ayarlanmışsa, sayfa bellekte *değil,* diskte bir yerdedir. Fiziksel bellekte olmayan bir sayfaya erişme eylemi genellikle **sayfa hatası** olarak adlandırılır.

BIR KENARA: TERMINOLOJININ VE DIĞER ŞEYLERIN DEĞIŞTIRILMESI

Sanal bellek sistemlerindeki terminoloji biraz kafa karıştırıcı olabilir ve makineler ve işletim sistemleri arasında farklılık gösterebilir. Örneğin, **sayfa hatası** daha genel olarak bir tür hata oluşturan bir sayfa tablosuna yapılan herhangi bir referansı ifade edebilir: bu, burada tartıştığımız hata türünü, yani sayfa mevcut değil hatasını içerebilir, ancak bazen yasadışı bellek erişimlerini de ifade edebilir. Aslında, kesinlikle yasal bir erişime (bir sürecin sanal adres alanına eşlenmiş, ancak o anda fiziksel bellekte olmayan bir sayfaya) "hata" dememiz gariptir; aslında buna **sayfa kaçırma** denmelidir. Ancak insanlar genellikle bir programın "sayfa hatası" yaptığını söylediklerinde, programın sanal adres alanının işletim sisteminin diske takas ettiği kısımlarına eriştiğini kastederler.

Bu davranışın "hata" olarak bilinmesinin nedeninin işletim sisteminin bunu ele alma mekanizmasından kaynaklandığını düşünüyoruz. Olağandışı bir şey olduğunda, yani donanımın nasıl başa çıkacağını bilmediği bir şey meydana geldiğinde, donanım, işleri daha iyi hale getirebileceğini umarak kontrolü işletim sistemine aktarır. Bu durumda, bir sürecin erişmek istediği bir sayfa bellekte eksiktir; donanım yapabileceği tek şeyi yapar, bu da bir istisna yaratmaktır ve işletim sistemi bundan sonrasını devralır. Bu, bir süreç yasadışı bir şey yaptığında olanlarla aynı olduğundan, bu faaliyeti "hata" olarak adlandırmamız belki de şaşırtıcı değildir.

Bir sayfa hatası üzerine, işletim sistemi sayfa hatasına hizmet etmesi için çağrılır. **Sayfa hatası** işleyicisi olarak bilinen belirli bir kod parçası çalışır ve şimdi açıkladığımız gibi sayfa hatasına hizmet etmelidir.

Sayfa Hatası

TLB kaçırmalarında iki tür sistemimiz olduğunu hatırlayın: donanım tarafından yönetilen TLB'ler (donanımın istenen çeviriyi bulmak için sayfa tablosuna baktığı sistem) ve yazılım tarafından yönetilen TLB'ler (bunu işletim sistemi yapar). Her iki tür sistemde de, bir sayfa mevcut değilse, sayfa hatasını ele almak için işletim sistemi görevlendirilir. Uygun şekilde adlandırılmış OS sayfa hatası **işleyicisi** ne yapılacağını belirlemek için çalışır. Neredeyse tüm sistemler sayfa hatalarını yazılımda ele alır; donanım tarafından yönetilen TLB'de bile donanım bu önemli görevi yönetmesi için işletim sistemine güvenir.

Bir sayfa mevcut değilse ve diske takas edilmişse, işletim sisteminin sayfa hatasını gidermek için sayfayı belleğe takas etmesi gerekecektir. Bu durumda bir soru ortaya çıkar: İşletim sistemi istenen sayfayı nerede bulacağını nasıl bilecektir? Birçok sistemde, sayfa tablosu bu tür bilgileri saklamak için doğal bir yerdir. Bu nedenle, işletim sistemi PTE'de normalde sayfanın PFN'si gibi veriler için kullanılan bitleri bir disk adresi için kullanabilir. İşletim sistemi bir sayfa için sayfa hatası aldığında, adresi bulmak için PTE'ye bakar ve sayfayı belleğe almak için diske istek gönderir.

Disk I/O tamamlandığında, işletim sistemi sayfa tablosunu güncelleyerek sayfayı mevcut olarak işaretler, sayfa tablosu girişinin (PTE) PFN alanını güncelleyerek yeni alınan sayfanın bellek içi konumunu kaydeder ve komutu yeniden dener. Bu sonraki deneme bir TLB ıskası yaratabilir, bu durumda TLB servis edilir ve çeviri ile güncellenir (bu adımdan kaçınmak için sayfa hatası servis edilirken TLB alternatif olarak güncellenebilir). Son olarak, son bir yeniden başlatma TLB'de çeviriyi bulacak ve böylece çevrilmiş fiziksel adreste bellekten istenen veriyi veya talimatı almaya devam edecektir.

G/Ç uçuş halindeyken sürecin **engellenmiş** durumda olacağını unutmayın durumuna geçer. Böylece, sayfa hatası servis edilirken işletim sistemi diğer hazır süreçleri çalıştırmakta özgür olacaktır. G/Ç pahalı olduğundan, bir sürecin G/Ç'si (sayfa hatası) ile diğerinin yürütülmesinin **üst üste binmesi,** çok programlı bir sistemin donanımını en etkili şekilde kullanmasının bir başka yoludur.

Ya Bellek Doluysa?

Yukarıda açıklanan işlemde, takas alanından bir sayfayı **yerleştirmek** için bol miktarda boş bellek olduğunu varsaydığımızı fark edebilirsiniz. Elbette durum böyle olmayabilir; bellek dolu olabilir (ya da buna yakın olabilir). Bu nedenle, işletim sistemi getirmek üzere olduğu yeni sayfa(lar)a yer açmak için önce bir ya da daha fazla sayfayı **çıkarmak** isteyebilir. Dışarı atılacak ya da **değiştirilecek** sayfayı seçme işlemi **sayfa değiştirme politikası** olarak bilinir.

Anlaşıldığı üzere, iyi bir sayfa değiştirme politikası oluşturmak için çok düşünülmüştür, çünkü yanlış sayfayı atmak program performansı üzerinde büyük bir maliyet oluşturabilir. 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 konudur; aslında bir sonraki bölümde yapacağımız şey de tam olarak budur. Şimdilik, burada açıklanan mekanizmaların üzerine inşa edilmiş böyle bir politikanın var olduğunu anlamak yeterlidir.

```
1VPN = (VirtualAddress & VPN_MASK) >> SHIFT
2    (Success, TlbEntry) = TLB_Lookup(VPN)
3if   (Success ==True)        //TLB Hit
4if        (CanAccess(TlbEntry.ProtectBits) == True)
5              Ofset     =VirtualAddress & OFFSET_MASK
6PhysAddr      = (TlbEntry.PFN << SHIFT) | Ofset
7Register      = AccessMemory(PhysAddr)
8else
9RaiseException     (PROTECTION_FAULT)
10   else                    //TLB Cevapsız
11PTEAddr    = PTBR + (VPN * sizeof(PTE))
12PTE        = AccessMemory(PTEAddr)
13if         (PTE.Valid == False)
14RaiseException     (SEGMENTATION_FAULT)
15else
16if              (CanAccess(PTE.ProtectBits) == False)
17RaiseException        (PROTECTION_FAULT)
18else            if (PTE.Present == True)
19//                   donanım tarafından yönetilen TLB varsayıldığında
20TLB_Insert           (VPN, PTE.PFN, PTE.ProtectBits)
21RetryInstruction     ()
22else            if (PTE.Present == False)
23RaiseException       (PAGE_FAULT)
```

Sayfa Arıza Kontrol Akış Algoritması (Donanım)

Sayfa Arıza Kontrol Akışı

Tüm bu bilgileri bir araya getirdiğimizde, artık bellek erişiminin tüm kontrol akışını kabaca çizebiliriz. Başka bir deyişle, birileri size "bir program bellekten bir veri aldığında 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 donanımın çeviri sırasında ne yaptığını, ikincisi ise işletim sisteminin bir sayfa hatası üzerine ne yaptığını göstermektedir.

Şekil 21.2'deki donanım kontrol akış diyagramından, bir TLB kaçırma durumu ortaya çıktığında anlaşılması gereken üç önemli durum olduğuna dikkat edin. Birincisi, sayfanın hem **mevcut** hem de **geçerli olduğu durumdur** (Satır 18-21); bu durumda, TLB hata işleyicisi PTE'den PFN'yi alabilir, komutu yeniden deneyebilir (bu sefer TLB isabetiyle sonuçlanır) ve böylece daha önce açıklandığı gibi (birçok kez) devam edebilir. İkinci durumda (Satır 22-23), sayfa hatası işleyicisi çalıştırılmalıdır; bu işlemin erişmesi için meşru bir sayfa olmasına rağmen (sonuçta geçerlidir), fiziksel bellekte mevcut değildir. Üçüncü (ve son olarak), erişim, örneğin programdaki bir hata nedeniyle geçersiz bir sayfaya olabilir (Satır 13-14). Bu durumda, PTE'deki diğer bitler gerçekten önemli değildir; donanım bu geçersiz erişimi yakalar ve işletim sistemi tuzak işleyicisi çalışarak muhtemelen hatalı süreci 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 alacağı

fiziksel bir çerçeve bulmalıdır; böyle bir sayfa yoksa, değiştirme algoritmasının çalışmasını ve bazı sayfaları bellekten atmasını beklememiz gerekir, böylece onları burada kullanmak için serbest bırakırız.

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

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

Elinde fiziksel bir çerçeve bulunan işleyici daha sonra sayfayı takas alanından 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 ıskası ile sonuçlanacak ve ardından başka bir yeniden deneme üzerine bir TLB isabeti olacak ve bu noktada donanım istenen öğeye erişebilecektir.

21.1 Değişimler Gerçekten Gerçekleştiğinde

Şimdiye kadar, değiştirmelerin nasıl gerçekleştiğini açıkladığımız şekilde, işletim sisteminin bellek tamamen dolana kadar beklediğini ve ancak o zaman başka bir sayfaya yer açmak için bir sayfayı değiştirdiğini (tahliye ettiğini) varsayıyoruz. Tahmin edebileceğiniz gibi, bu biraz gerçekçi değildir ve işletim sisteminin belleğin küçük bir bölümünü daha proaktif bir şekilde boş tutması için birçok neden vardır.

Az miktarda belleği boş tutmak için çoğu işletim sistemi, sayfaları bellekten ne zaman boşaltmaya başlayacağına karar vermeye yardımcı olmak için bir tür **yüksek filigran (**HW) ve **düşük filigran** (LW) kullanır. Bunun işleyişi şu şekildedir: işletim sistemi LW'den daha az boş sayfa olduğunu fark ettiğinde, belleği boşaltmaktan sorumlu bir arka plan iş parçacığı çalışır. İş parçacığı, kullanılabilir HW sayfaları olana kadar sayfaları tahliye eder. Bazen **takas daemon'u** ya **da sayfa daemon**'u[1] olarak adlandırılan arka plan iş parçacığı, daha sonra çalışan işlemler ve işletim sisteminin kullanması için bir miktar bellek serbest bıraktığı için mutlu bir şekilde uykuya geçer.

Bir kerede çok sayıda değiştirme gerçekleştirerek yeni performans optimizasyonları mümkün hale gelir. Örneğin, birçok sistem bir dizi sayfayı **kümeler** veya **gruplandırır** 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 ek yüklerini azaltır ve böylece performansı belirgin şekilde artırır. Arka plandaki disk belleği iş parçacığı ile çalışmak için, Şekildeki kontrol akışı
21.3 biraz değiştirilmelidir; doğrudan bir değiştirme yapmak yerine, algoritma bunun yerine mevcut boş sayfa olup olmadığını kontrol edecektir.

Eğer yoksa, arka plandaki sayfalama iş parçacığına boş sayfalara ihtiyaç olduğunu bildirir; iş parçacığı bazı sayfaları serbest bıraktığında, orijinal iş parçacığını yeniden uyandırır ve bu iş parçacığı istenen sayfayı sayfalayıp işine devam edebilir.

[1]Genellikle "şeytan" olarak telaffuz edilen "daemon" kelimesi, yararlı bir şey yapan bir arka plan iş parçacığı veya süreci için kullanılan eski bir terimdir. Terimin kaynağının Multics [CS94] olduğu ortaya çıktı (bir kez daha!).

İPUCU: ARKA PLANDA ÇALIŞIN

Yapmanız gereken bazı işler olduğunda, verimliliği artırmak ve işlemlerin gruplandırılmasına izin vermek için bu işleri **arka planda** yapmak genellikle iyi bir fikirdir. İşletim sistemleri genellikle arka planda çalışır; örneğin, birçok sistem verileri diske yazmadan önce dosya yazımlarını bellekte arabelleğe alır. Bunu yapmanın birçok olası faydası vardır: disk artık aynı anda birçok yazma alabileceği ve böylece bunları daha iyi programlayabileceği için disk verimliliğinin artması; uygulama yazmaların oldukça hızlı bir şekilde tamamlandığını düşündüğü için yazmaların gecikme süresinin iyileşmesi; yazmaların diske hiç gitmemesi gerekebileceği için (örneğin, dosya silinirse) iş azaltma olasılığı; ve arka plan çalışması muhtemelen sistem boştayken yapılabileceği için **boşta kalma süresinin daha iyi kullanılması,** böylece donanımın daha iyi kullanılması [G+95].

21.7 Özet

Bu kısa bölümde, bir sistem içinde fiziksel olarak mevcut olandan daha fazla belleğe erişme kavramını tanıttık. Bunu yapmak için sayfa tablosu yapılarında daha fazla karmaşıklık gerekir, çünkü sayfanın bellekte mevcut olup olmadığını bize bildirmek için bir **mevcut biti** (bir tür) dahil edilmelidir. Olmadığında, işletim sistemi **sayfa hatası işleyicisi sayfa hatasına** hizmet etmek için çalışır ve böylece istenen sayfanın diskten belleğe aktarılmasını ayarlar, belki de önce bellekteki bazı sayfaları değiştirerek yakında takas edilecek olanlara yer açar.

Önemli (ve şaşırtıcı!) olarak, tüm bu işlemlerin süreç için **şeffaf bir şekilde** gerçekleştiğini hatırlayın. Süreç söz konusu olduğunda, sadece kendi özel, bitişik sanal belleğine erişmektedir. Perde arkasında, sayfalar fiziksel bellekte rastgele (bitişik olmayan) konumlara yerleştirilir ve bazen bellekte bile bulunmazlar, diskten getirilmeleri gerekir. Genel durumda bir bellek erişiminin hızlı olmasını umarken, bazı durumlarda bunu sağlamak için birden fazla disk işlemi gerekebilir; tek bir komutu gerçekleştirmek kadar basit bir işlemin tamamlanması en kötü durumda milisaniyeler sürebilir.

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!

Ev Ödevi (Ölçme)

Bu ödev size yeni bir araç olan **vmstat**'ı ve bellek, CPU ve I/O kullanımını anlamak için nasıl kullanılabileceğini tanıtmaktadır. Aşağıdaki alıştırmalara ve sorulara geçmeden önce ekteki README'yi okuyun ve `mem.c`'deki kodu inceleyin.

Sorular

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

Şimdi, bir pencerede, her saniye makine kullanımı hakkında istatistikler gösteren `vmstat 1`'i çalıştırın. Çıktısını anlayabilmek için man sayfasını, ilgili README'yi ve ihtiyacınız olan diğer bilgileri okuyun. Aşağıdaki alıştırmaların geri kalanı için `vmstat`'ı çalıştıran bu pencereyi bırakın.

Şimdi, `mem.c` programını çok az bellek kullanarak çalıştıracağız. Bu, `./mem 1` (sadece 1 MB bellek kullanır) yazarak gerçekleştirilebilir. `mem` çalıştırıldığında CPU kullanım istatistikleri nasıl değişiyor? `Kullanıcı zamanı` sütunundaki sayılar mantıklı mı? Aynı anda birden fazla `mem` örneği çalıştırıldığında bu durum nasıl değişir?

1 Cevap: vmstat 1Linux veya macOS gibi Unix benzeri bir işletim sisteminde sanal bellek, CPU kullanımı ve diğer sistem istatistikleri hakkında bilgi görüntülemek için kullanılabilen bir komuttur. 1Seçeneği ile çalıştırıldığında vmstat, istatistikleri saniyede bir kez görüntüler. vmstat komutu, sistemde bulunan RAM, takas alanı, CPU ve I/O (G/Ç) gibi çeşitli istatistikleri gösterir. Bu istatistikleri anlamak için, man sayfasını veya ilgili README dosyasını okumalıyız.

Eğer mem programı az miktarda bellek kullanarak çalıştırılırsa, CPU kullanım istatistikleri de az olacaktır. Örneğin, mem programı sadece 1 MB bellek kullanarak çalıştırıldığında, CPU kullanım istatistikleri düşük olacaktır. Kullanıcı zamanı sütunundaki sayılar da düşük olacaktır, çünkü program az miktarda işlem gerçekleştirecektir.Aynı anda birden fazla mem örneği çalıştırıldığında ise, CPU kullanım istatistikleri artacaktır. Bu, çalışan örneklerin işlemlerini gerçekleştirirken daha fazla işlemci gücüne ihtiyaç duyulması anlamına gelecektir. Kullanıcı zamanı sütunundaki sayılar da artacaktır.

Komutun çıktısı vmstat 1aşağıdaki alanları içerir:

r: CPU zamanını bekleyen işlemlerin sayısı.
b: Kesintisiz uyku durumunda olan işlemlerin sayısı.
swpd: Kilobayt cinsinden kullanılmakta olan sanal bellek miktarı.
free: Boş bellek miktarı, kilobayt cinsinden.
buff: Kilobayt cinsinden arabellek olarak kullanılan bellek miktarı.
cache: Kilobayt cinsinden önbellek olarak kullanılan bellek miktarı.
si: Saniyede diskten takas edilen bellek miktarı, kilobayt cinsinden.
so: Kilobayt cinsinden saniyede diske değiştirilen bellek miktarı.
bi: Saniyede bir blok cihazından alınan blok sayısı.
bo: Saniyede bir blok cihazına gönderilen blok sayısı.
in: Hem donanım hem de yazılım kesintileri dahil, saniyedeki kesinti sayısı.
cs: Saniyedeki içerik değiştirme sayısı.
us: Kullanıcı işlemlerinde harcanan CPU zamanının yüzdesi.
sy: Sistem süreçlerinde harcanan CPU zamanının yüzdesi.
id: Boşta kalan CPU süresinin yüzdesi.
wa: G/Ç için bekleyen CPU süresinin yüzdesi.
st: Bir sanal makineden çalınan CPU zamanının yüzdesi.

Vmstat Komutu, sisteminizin performansını izlemek ve CPU veya bellek darboğazları gibi olası sorunları belirlemek için kullanabilirsiniz .Komutu çalıştırdığımızda **`vmstat 1`**, her saniye CPU, bellek ve disk kullanımı gibi makine kullanımıyla ilgili istatistikleri görüntüler. **`Vmstat`** Komut, bir sistemin performansını izlemek ve olası sorunları belirlemek için yararlı olabilir .Komutun çıktısı **`vmstat`**, CPU tarafından kullanılan kullanıcı ve sistem süresi, boş ve kullanılan bellek miktarı ve diske okunan ve yazılan blok sayısı gibi çeşitli istatistikler için sütunlar içerir. **`Vmstat`** Her sütunun anlamı ve mevcut seçenekler hakkında daha fazla bilgi edinmek için man sayfasını okuyabilir **`../mem`**

1Yalnızca 1 MB bellek kullanan komutu çalıştırdığınızda , **vmstat** çıktıdaki CPU kullanım istatistiklerinin değiştiğini fark edebiliriz. İşlemleri kullanıcı modunda çalıştırmak için harcanan CPU zamanı miktarını gösteren sütun, program CPU'yu kullandıkça **us** artabilir . **mem** İşlemleri **sy** çekirdek modunda çalıştırmak için harcanan CPU süresini gösteren sütun, işletim sisteminin **mem** programla ilgili herhangi bir görevi gerçekleştirmesi gerektiğinde de artabilir.Kullanıcı zamanı sütunundaki sayılar, program bağlamında **mem** ve kullandığı bellek miktarında anlamlı olmalıdır. Program yalnızca az miktarda bellek kullandığından , **mem** CPU kullanımı önemli olmayabilir ve kullanıcı zamanı sütunundaki sayılar nispeten düşük olabilir.Programın birden fazla örneğini **mem** aynı anda çalıştırır, **vmstat** çıktıdaki CPU kullanım istatistiklerinin buna göre değiştiğini fark edebilir. Programın ek örnekleri CPU'yu kullandıkça **us** ve **sy** sütunları artabilir ve bu da **mem** genel olarak daha yüksek CPU kullanımına neden olur. **Mem** CPU kullanım istatistiklerindeki kesin değişiklik, belirli bir sisteme ve programın çalışan örnek sayısına bağlı olacaktır .

2: Şimdi `mem'i` çalıştırırken bazı bellek istatistiklerine bakmaya başlayalım. İki sütuna odaklanacağız: `swpd` (kullanılan sanal bellek miktarı) ve `free` (boşta kalan bellek miktarı). . `/mem` 1024 (1024 MB ayırır) çalıştırın ve bu değerlerin nasıl değiştiğini izleyin. Sonra çalışan programı sonlandirin (control-c yazarak) ve değerlerin nasıl değiştiğini tekrar izleyin. Değerler hakkında ne fark ettiniz? Özellikle, programdan çıkıldığında `boş` sütun nasıl değişiyor? `Mem` çıktığında boş bellek miktarı beklenen miktarda artıyor mu?

2: Cevap: `./mem 1024` 1024 MB bellek ayıran komutu çalıştırdığımızda, komut çıktısındaki **swpd** sütunun programın **top** ayırdığı bellek miktarı kadar arttığını fark edebilirsiniz. **Mem** Bu, programın, işletim sistemi tarafından yönetilen ve gerekirse diskle takas edilebilen ve diskten değiştirilebilen sanal bellek kullandığını gösterir.Program çalışırken komut çıktısındaki **free** sütun **top** da değişebilir . **mem** Bu sütun, boşta kalan veya şu anda herhangi bir program tarafından kullanılmayan belleği gösterir. Program hafıza kullandığından , **mem** boşta kalan hafıza miktarı azalabilir.Çalışan **mem** programı tuşuna basarak sonlandırdığımızda , sütunun program tarafından ayrılan bellek miktarı kadar azaldığını **Ctrl+C** fark edebilirsiniz . Bu, programın artık sanal bellek kullanmadığını gösterir. Daha önce program tarafından kullanılan bellek artık diğer programların kullanımına açık olduğundan, sütun da artabilir. **swpd mem free mem mem** Belleğin bir kısmı işletim sistemi veya diğer programlar tarafından kullanılabileceğinden , boş bellek miktarı program tarafından ayrılan tam bellek miktarı kadar artmayabilir . **mem** Ancak, programdan çıkıldığında boş bellek miktarında önemli bir artış görmemiz gerekir Eğer mem programı çalıştırıldığımızda, swpd sütunu 1024 MB'lık bir artış gösterir ve free sütunu 1024 MB azalır. Bu, mem programının RAM'de yer kapladığı anlamına gelir. Programdan çıkıldığında ise, swpd sütunu tekrar eski değerine döner ve free sütunu tekrar 1024 MB artar. Bu, mem programının RAM'den çıkarıldığı anlamına gelir ve boş bellek miktarı beklenen miktarda artar.

Not: Bu sonuçların özellikle RAM miktarı ve çalışan diğer programlar gibi diğer faktörler de etkileyebilecek.

3: Daha sonra diske ve diskten ne kadar `takas` yapıldığını gösteren takas sütunlarına (`si` ve `so`) bakacağız. Elbette, bunları etkinleştirmek için `mem'i` büyük miktarda bellekle çalıştırmanız gerekecektir. İlk olarak, Linux sisteminizde ne kadar boş bellek olduğunu inceleyin (örneğin, `cat /proc/meminfo`; `/proc` dosya sistemi ve burada bulabileceğiniz bilgi türleri hakkında ayrıntılar için `man proc` yazın). İlk girdilerden biri `/proc/meminfo` sisteminizdeki toplam bellek miktarıdır. Bunun 8 GB bellek gibi bir şey olduğunu varsayalım; eğer öyleyse, `mem 4000` (yaklaşık 4 GB) çalıştırarak ve takas giriş/çıkış sütunlarını izleyerek başlayın. Hiç sıfır olmayan değerler

veriyorlar mı? Sonra `5000`, `6000`, vb. ile deneyin. Program ikinci döngüye (ve ötesine) girdiğinde, ilk döngüye kıyasla bu değerlere ne oluyor? İkinci, üçüncü ve sonraki döngüler sırasında ne kadar veri (toplam) içeri ve dışarı değiştirilir? (sayılar mantıklı mı?)

3 Cevap: Komuttaki **`si`** ve **`so`** sütunları **`top`**, diske ve diskten takas edilen veri miktarını gösterir. Bu, sistemde meydana gelen ve performansı etkileyebilecek takas miktarını izlemek için yararlı olabilir. Bir Linux sistemindeki boş bellek miktarını belirlemek için **`cat /proc/meminfo`**komutu kullanabilir. Bu komut, toplam bellek miktarı ve boş bellek miktarı da dahil olmak üzere sistem belleği hakkında çeşitli bilgileri yazdırır. Örneğin, sistem 8 GB belleğe sahipse, komutun çıktısı şöyle görünebilir

```
MemTotal:        8192 MB
MemFree:         4096 MB
```

Bu örnekte, sistem 8 GB toplam belleğe ve 4 GB boş belleğe sahiptir. **`Mem`** Programı 4 GB (4000 MB) gibi büyük bir bellekle çalıştırırsanız , **`si`** ve **`so`** sütunlarında sıfırdan farklı değerler görebilirsiniz. Bu, bazı verilerin diske ve diskten değiştirildiğini gösterir. Eğer bir Linux sisteminde /proc/meminfo dosyasını inceliyorsak, orada toplam fiziksel bellek miktarınızı görebilirsiniz. Örneğin, "MemTotal" satırı toplam fiziksel bellek miktarını gösterir. Eğer sisteminizde 4 GB bellek varsa ve mem komutunu 4000 (yaklaşık 4 GB) değerine çalıştırırız, programın ilk döngüsünde si ve so sütunlarına bakarsanız, bu sütunlarda hiç sıfır olmayan değerler olup olmadığını görebilirsiniz. Eğer ikinci, üçüncü ve sonraki döngüler sırasında mem komutunu artırırsak (örneğin, 5000, 6000, vb.), ikinci, üçüncü ve sonraki döngülerde si ve so sütunlarında değerlerin değişip değişmediğini görebiliriz. Bu sütunlar, programın çalışırken diskten takas yaptığını gösterir. Eğer program ikinci, üçüncü ve sonraki döngülerde daha fazla takas yapıyorsa, bu genellikle programın daha fazla bellek gereksinimine işaret eder ve programın daha fazla bellek kullanımına işaret edebilir. Bu durumda, mem komutunu daha yüksek değerlere çalıştırarak programın bellek kullanımını izleyebiliriz. Ancak, dikkat etmeniz gereken bir nokta var: Eğer sisteminizde yeterli bellek yoksa, programın çalışması sırasında takas sıklığı artacak ve bu da sistem performansını olumsuz etkileyebilir. Bu nedenle, mem komutunu çalıştırırken sistemdeki boş bellek miktarını da dikkate almalıyız.

4: Yukarıdaki gibi aynı deneyleri yapın, ancak şimdi diğer istatistikleri izleyin (CPU kullanımı ve blok G/Ç istatistikleri gibi). `Mem` çalışırken nasıl değişiyorlar?

4Cevap : mem komut, çok sayıda yoğun bellek gerektiren işlemler gerçekleştirmesine neden olacak şekilde kullanılıyorsa, bu istatistikleri etkilemesi olasıdır. Örneğin, eğermem komutu bir dizi bellek testi veya kıyaslama gerçekleştirmek için kullanılıyorsa, potansiyel olarak CPU kullanımında artışa neden olabilir veya G/Ç etkinliğini engelleyebilir. Komut , mem belleği boşaltmak veya bellekle ilgili diğer görevleri gerçekleştirmek için kullanılıyorsa, CPU kullanımında geçici bir düşüşe neden olabilir veya G/Ç etkinliğini engelleyebilir, Mem programı çalışırken, CPU kullanımı ve blok G/Ç (block I/O) istatistikleri değişebilir. CPU kullanımı, mem programının işlemleri gerçekleştirirken kullandığı işlemci gücünü gösterir ve genellikle yüksek olabilir. Blok G/Ç istatistikleri ise, diske yapılan veri okuma ve yazma işlemlerini gösterir. Örneğin, mem programı verileri RAM'den disk'e yazdığında veya disk'ten RAM'e okuduğunda, blok G/Ç istatistikleri değişebilir.

Not: Bu sonuçların özellikle çalışan diğer programlar gibi diğer faktörler de etkileyebilecek.

5: Şimdi performansı inceleyelim. `Mem` için belleğe rahatça sığabilecek bir girdi seçin (sistemdeki bellek miktarı 8 GB ise `4000` diyelim). Döngü 0 ne kadar sürüyor (ve sonraki döngüler 1, 2, vb.)? Şimdi bellek boyutunun rahatça ötesinde bir boyut seçin (8 GB bellek olduğunu varsayarak yine `12000 diyelim`) . bellek). Burada döngüler ne kadar sürüyor? Bant genişliği sayıları

nasıl karşılaştırılır? Sürekli takas yaparken ve her şeyi rahatça belleğe sığdırırken performans ne kadar farklıdır? `Mem` tarafından kullanılan bellek boyutu x ekseninde ve söz konusu belleğe erişim bant genişliği y ekseninde olacak şekilde bir grafik oluşturabilir misiniz? Son olarak, hem her şeyin belleğe sığdığı hem de sığmadığı durumlar için ilk döngünün performansı sonraki döngülerinkiyle nasıl karşılaştırılır?

5 Cevap :. Bir döngünün mem komutu kullanarak çalışması için gereken süre, girdinin boyutu, komutla kullanılan belirli parametreler mem ve bilgisayarın performansı gibi bir dizi faktöre bağlı olarak değişebilir. Bununla birlikte, genel olarak, mem komut nispeten hafif bir yardımcı programdır ve büyük girdiler için bile çalıştırılması önemli miktarda zaman almamalıdır. Genel olarak, bir döngünün yürütülmesi için gereken süre, girdinin boyutu, kullanılabilir bellek miktarı, işlemcinin hızı ve kodun verimliliği gibi çeşitli faktörlere bağlı olacaktır. Girdi boyutu kullanılabilir bellek içinde rahat bir şekildeyse, sistem bellekteki verilere bellekte girip çıkmak zorunda kalmadan erişebileceğinden, döngü nispeten hızlı bir şekilde yürütülebilmelidir. Bu, nispeten yüksek bant genişliği ve iyi performansla sonuçlanmalıdır. Öte yandan, giriş boyutu kullanılabilir bellekten daha büyükse, sistem sürekli olarak belleğin içindeki ve dışındaki verileri değiş tokuş etmek zorunda kalacaktır, bu da zaman alıcı olabilir ve döngünün performansını yavaşlatabilir. Bu, sistemin verilere erişilebilmesi için verileri belleğe yüklemesi gerekeceğinden, özellikle ilk döngü için daha düşük bant genişliğine ve daha kötü performansa neden olacaktır.

Sistem, birinci döngüden sonra bellekteki verilere erişimini optimize edebildiğinden, birinci döngünün performansı sonraki döngülerden farklı olabilir. Mem programı bellek boyutunun rahatça ötesinde bir boyut seçildiğinde, programın performansı düşecektir. Bu, sistem tarafından takas yapılmasına neden olacaktır. Takas, RAM'de bulunmayan verilerin disk üzerinde saklanması anlamına gelir ve RAM'de yer açıldığında, veriler RAM'e geri yüklenir. Bu, performansı yavaşlatacaktır, çünkü veriler disk üzerinden RAM'e okunması ve yazılması daha yavaştır.

Aşağıda, bellek boyutunun artışıyla birlikte döngülerin süresini gösteren bir grafik :

Bellek boyutu	İlk Döngü Süresi	Sonraki Döngüler Süresi
4000	X	Y
12000	Z	W

X ve Y değerleri, bellek boyutu 4000 olarak seçildiğinde ilk döngü ve sonraki döngülerin sürelerini, Z ve W değerleri ise bellek boyutu 12000 olarak seçildiğinde ilk döngü ve sonraki döngülerin sürelerini gösterir. X ve Z değerleri, bellek boyutunun artışıyla birlikte döngülerin süresinin artacağını gösterirken, Y ve W değerleri ise sonraki döngülerin ilk döngülerden daha hızlı olacağını gösterir. Sonuç olarak, bellek boyutunun rahatça ötesinde bir boyut seçildiğinde, döngülerin süresi artacak ve ilk döngüler daha yavaş olacaktır. Ancak sonraki döngüler, ilk döngülerden daha hızlı olacaktır, çünkü veriler zaten RAM'de mevcuttur ve takas yapılması gerekmemektedir.

6: Takas alanı sonsuz değildir. Ne kadar takas alanı olduğunu görmek için `-s` bayrağı ile `swapon` aracını kullanabilirsiniz. Eğer `mem`'i giderek daha büyük değerlerle çalıştırmaya çalışırsanız, takas alanında mevcut görünenin ötesinde ne olur? Bellek tahsisi hangi noktada başarısız olur?

6 Cevap: Takas alanı, sabit diskin, işletim sistemi tarafından şu anda bellekteki programlar tarafından kullanılmayan verileri depolamak için kullanılan bir bölümüdür. Sistemdeki fiziksel bellek (RAM) miktarı, programlar tarafından kullanılan tüm verileri depolamak için yeterli değilse, bu yararlı olabilir. Ancak takas alanı sonsuz değildir ve takasta depolanabilecek veri miktarının bir sınırı vardır. **`swapon -s`** Sistemde ne kadar takas alanı olduğunu görmek için komutu kullanabilirsiniz . Bu komut, toplam boyut, kullanılan boyut ve kalan boyut dahil olmak üzere

kullanılabilir takas alanı hakkında bilgi görüntüler. Programı, takasta mevcut görünenin ötesinde, giderek daha büyük değerlerle çalıştırmaya çalışırsak `mem`, sistemde takas alanı bittiğinde bir hatayla karşılaşabiliriz. Bu, program tarafından ayrılan veri miktarı `mem` sistemdeki kullanılabilir takas alanını aşarsa meydana gelebilir. Bu noktada, bellek ayırma işlemi başarısız olur ve `mem` program, istenen miktarda bellek ayıramaz. Bellek tahsisinin başarısız olduğu tam nokta, belirli bir sisteme ve kullanılabilir takas alanı miktarına bağlı olacaktırTakas alanı, sistem tarafından kullanılan sanal bellek (swap) miktarını gösterir. Takas alanı, RAM'de bulunmayan verilerin disk üzerinde saklanmasını sağlar ve RAM'de yer açıldığında, veriler RAM'e geri yüklenir.

Eğer mem programı takas alanının ötesine çıkmaya çalışırsa, bellek tahsisi başarısız olacaktır. Bu durumda, program hata mesajı vererek çalışmayı durduracaktır. Programın kullandığı bellek miktarı, takas alanındaki mevcut görünenin ötesine çıktığında bellek tahsisi başarısız olacaktır. Takas alanının ne kadar olduğunu görmek için swapon aracını -s bayrağı ile çalıştırabiliriz. Bu, sistemde kullanılan takas alanını gösterir ve takas alanının ne kadarı kullanıldığını gösterir. Eğer takas alanı tamamen dolduysa ve program daha fazla bellek tahsisi yapmaya çalışırsa, bellek tahsisi başarısız olacaktır. Takas alanının ne kadar olduğu, sistemde bulunan disk miktarına ve yapılandırmaya göre değişebilir. Takas alanını büyütmek için, sistemde bulunan boş disk alanını takas alanı olarak ayırabilirsiz

7: Son olarak, eğer ileri düzeydeyseniz, sisteminizi `swapon` ve `swapoff` kullanarak farklı takas aygıtları kullanacak şekilde yapılandırabilirsiniz. Ayrıntılar için man sayfalarını okuyun. Farklı donanımlara erişiminiz varsa, klasik bir sabit sürücüye, flash tabanlı bir SSD'ye ve hatta bir RAID dizisine takas yaparken takas ping performansının nasıl değiştiğini görün. Daha yeni cihazlarla takas performansı ne kadar artırılabilir? Bellek içi performansa ne kadar yaklaşabilirsiniz?

7 Cevap : swapon ve swapoff komutları, sistemde bulunan takas alanını açma ve kapatma işlemlerini yapar. Bu komutları kullanarak, sistemde bulunan farklı takas aygıtlarını kullanarak takas yapılmasını yapılandırabiliriz. Örneğin, swapon komutu ile bir flash tabanlı SSD veya bir RAID dizisi gibi farklı donanımları takas alanı olarak ayarlayabiliriz. Bu sayede, takas performansını arttırabilir ve bellek içi performansa daha da yaklaşabiliriz.

Takas performansını ölçmek için, takas alanını kullanarak bellek tahsisi yapılan bir program çalıştırabilir ve programın performansını ölçebilir. Örneğin, mem programını farklı takas aygıtlarını kullanarak çalıştırıp döngülerin süresini ölçerek takas performansını karşılaştırabiliriz. Böylelikle, farklı takas aygıtlarının takas performansı arasındaki farkı görebiliriz.

Not: Takas performansını arttırmak için, sistemde bulunan takas alanının yeterli miktarda olması ve takas alanının hızlı bir aygıt üzerinde olması önemlidir. Takas alanını büyütmek için, sistemde bulunan boş disk alanını takas alanı olarak ayırabiliriz. Takas alanını hızlı bir aygıt üzerinde tutmak için ise, flash tabanlı SSD veya RAID dizisi gibi hızlı aygıtları kullanabiliriz.