Fiziksel Belleğin ötesinde:İlkeler

Bir sanal bellek yöneticisinde, çok fazla boş bellek olduğunda hayat çok kolaydır. Bir sayfa hatası oluşur, boş sayfa listesinde boş bir sayfa bulur ve bunu hatalı sayfaya atarsınız. Merhaba, İşletim Sistemi, tebrikler! Bunu tekrar yaptınız.

Ne yazık ki, küçük hafıza boşken işler biraz daha ilgi çekici hale geliyor. Böyle bir durumda**, bu bellek basıncı**, etkin olarak kullanılan sayfalara yer açmak için işletim sistemini sayfalar arasında geçiş yapmaya zorlar. **Hangi sayfanın (veya sayfaların) işletim** sisteminin değiştirme ilkesine dahil edildiğine karar vermek; eski sistemlerde çok az fiziksel bellek olduğu için, geçmişte eski sanal bellek sistemlerinin verdiği en önemli kararlardan biriydi. En azından bu, biraz daha fazla bilinmeye değer ilginç bir politika kümasıdır. Ve bu nedenle sorunumuz:

HAM MADDE: HANGI SAYFANIN TAHLIYE EDILECEĞINE NASIL KARAR VERIRSINIZ

İşletim sistemi hangi sayfanın (veya sayfaların) bellekten tahliye edileceğine nasıl karar verebilir? Bu karar, genellikle bazı genel prensipleri (aşağıda ele alınmıştır) takip eden ancak aynı zamanda köşe olayı davranışlarından kaçınmak için belirli ince ayarlar içeren sistemin değiştirme politikasıyla verilir.

22.1 Önbellek Yönetimi

Politikalara dalmadan önce, ilk olarak çözmeye çalıştığımız sorunu daha ayrıntılı olarak açıklıyoruz. Ana belleğin sistemdeki tüm sayfaların bir kısmını içermesi göz önüne alındığında, sistemdeki sanal bellek sayfaları için önbellek olarak doğru bir şekilde görüntülenebilir. Bu nedenle, bu önbellek için bir değiştirme ilkesi seçmenin amacı**, önbellekteki hata sayısını en aza indirmek**, yani diskten bir sayfa alma sayısını en aza indirmektir. Alternatif olarak, hedeflerimizi **önbellek vuruşlarının sayısını en üst düzeye çıkarmak,** yani erişilecek bir sayfanın bellekte kaç kez bulunduğunu görmek olarak da görebiliriz.

Önbellek vuruşlarının ve atlamalarının sayısını bilmek, bir programın ortalama bellek erişim süresini (AMAT) hesaplamamıza izin verir (metrik bir bilgisayar, donanım önbellekleri için bilgi işlem mimarları [HP06]). Özellikle, bu değerler göz önüne alındığında, bir programın AMAT değerini aşağıdaki gibi hesaplayabiliriz:

$$AMAT = T_M + (P_{MISS} \cdot T_D)$$

TM, belleğe erişim maliyetini, TD diske erişim maliyetini ve PMISS 'yi önbellekteki verileri bulamama olasılığını (bir eksik) temsil eder; PMISS 0.0 ile 1.0 arasında değişiklik gösterir ve bazen olasılık yerine bir yüzde eksiklik oranı ifade ederiz (örn. %10'lik bir eksiklik oranı PMISS =0.10 anlamına gelir). Not bellekteki verilere erişim maliyetini her zaman ödersiniz; ancak, 'yi kaçırdığınızda, verileri diskten alma maliyetini de ödemeniz gerekir.

Örneğin, (küçük) adres alanına sahip bir makine düşünelim:
256 baytlık sayfalarla 4KB. Bu nedenle, sanal bir adresin iki bileşeni vardır: 4 bit VPN (en önemli bitler) ve 8 bit ofset (en önemsiz bitler). Bu nedenle, bu örnekteki bir işlem toplam 24 veya 16 sanal sayfaya erişebilir. Bu örnekte, işlem aşağıdaki bellek referanslarını oluşturur (sanal adresler gibi): 0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700, 0x800, 0x900. Bu sanal adresler, adres alanının ilk on sayfasının her birinin ilk baytını ifade eder (sayfa numarası her sanal adresin ilk onaltılık basamağıdır).

Sanal sayfa 3 dışında her sayfanın zaten bellekte olduğunu varsayalım. Bu nedenle, bellek referansı dizimiz şu davranışla karşılaşır: Hit, hit, miss, hit, hit, hit, vur, vur, vur, vur. 10 referansın 9'si bellekte olduğundan, darbe oranını (bellekte bulunan referansların yüzdesi) %90 hesaplayabiliriz. **Bu** nedenle, eksik oranı %10'dir (PMISS =0.1). Genel olarak PHIT + PMISS = 1.0; vurma oranı artı kaçırma oranı toplamı %100'e kadar.

AMAT'yi hesaplamak için belleğe erişim maliyetini ve diske erişim maliyetini bilmemiz gerekir. Belleğe (TM) erişim maliyetinin yaklaşık 100· 1.0001 nanosaniye olduğunu ve diske (TD) erişim maliyetinin yaklaşık 10 milisaniye olduğunu varsayarak şu AMAT'ye sahibiz: 100NS + 0.1 10MS, YANI 100NS + 1MS veya yaklaşık 1 milisaniye. Vurma hızımız %99.9 (PMISS=0.001) olsaydı sonuç oldukça farklıdır: AMAT 10.1 mikrosaniye veya yaklaşık 100 kat daha hızlıdır. Vurma oranı %100'e yaklaştıkça AMAT 100 nanosaniye yaklaşır.

Ne yazık ki, bu örnekte de gördüğünüz gibi, modern sistemlerde disk erişim maliyeti o kadar yüksektir ki, küçük bir kayıp oranı bile çalışan programların genel AMAT'sine hakim olacaktır. Açık bir şekilde, disk hızında olabildiğince çok hatalı veya yavaş çalışmalıyız. Bu konuda yardımcı olmak için şu anda olduğu gibi dikkatli bir şekilde akıllı bir politika geliştirmemiz gerekir.

22.2 Optimum Değiştirme İlkesi

Belirli bir değiştirme politikasının nasıl çalıştığını daha iyi anlamak için bunu mümkün olan en iyi değiştirme politikasıyla karşılaştırmak iyi olacaktır. Ortaya çıkan bu **gibi bir optimum** politika, uzun yıllar önce Belady tarafından geliştirildi [B66] (ilk olarak BU POLITIKAYI MIN olarak adlandırdı). Optimum değiştirme politikası, genel olarak en az sayıda hatalı duruma yol açar. Belady, basit olduğunu gösterdi (ama ne yazık ki uygulanması zor!) *gelecekte en çok erişilecek sayfanın yerini alan yaklaşım*, en düşük olası önbellek hatalarına yol açan en iyi ilkedir.

İPUCU: OPTIMAL ILE KARŞILAŞTIRMA YARARLIDIR

Optimal gerçek bir politika olarak çok pratik olmasa da, simülasyon veya diğer etütlerde karşılaştırma noktası olarak inanılmaz derecede kullanışlıdır. Yeni algoritmanın %80'lik bir vuruş oranına sahip olduğunu söylemek, izolasyon anlamında bir anlam ifade etmedi; optimal'in %82'lik bir darbe oranına (ve dolayısıyla yeni yaklaşımınıza) ulaştığını söylemek
optimal'e oldukça yakındır) sonucu daha anlamlı hale getirir ve bağlam verir. Bu nedenle, gerçekleştirdiğiniz tüm çalışmalarda, en uygun şeyin ne olduğunu bilmek daha iyi bir karşılaştırma yapmanızı, ne kadar gelişmenin hala mümkün olduğunu göstermenizi ve ideal[AD03] için yeterince yakın olduğu için politikanızı daha iyi hale getirmeyi ne zaman bırakabileceğinizi bilmenizi sağlar.

Umarım, optimum politikanın ardındaki entüsyon mantıklı olur. Bunu şöyle düşünün: Bir sayfa atmanız gerekiyorsa, neden şimdi en çok ihtiyaç duyulan sayfayı atmıyorsunuz? Bunu yaparak, önbellekteki diğer tüm sayfaların en uzak olandan daha önemli olduğunu söyleyeceksin. Bunun doğru olmasının nedeni çok basit: En uzaktaki sayfaya bakmadan önce diğer sayfalara başvuracaksınız.

Optimum politikanın verdiği kararları anlamak için basit bir örneği inceleyelim. Bir programın aşağıdaki sanal sayfa akışına eriştiğini varsayın: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1.Şekil 22.1, üç sayfaya sığan bir önbellek varsayılarak optimal davranışı göstermektedir.

Şekilde aşağıdaki işlemleri görebilirsiniz. Önbelleğin boş durumda başlaması, ilk üç erişinin de hatalı olması şaşırtıcı değildir; böyle bir eksiklik bazen **soğuk çalıştırma kaçırması (**veya **zorunlu kaçırma**) olarak da adlandırılır. Ardından, her ikisi de önbellekte yer alan 0. Ve 1. Sayfalara tekrar atıfta bulunuyoruz. Son olarak, başka bir eksiklik daha yaşıyoruz (3. Sayfaya kadar), ancak bu sefer önbellek dolmuştur; değiştirme işlemi yapılmalıdır! Hangi soru soruluyor: Hangi sayfayı değiştirmeliyiz? En iyi ilke ile önbellekte bulunan her sayfa için geleceği inceliyoruz (0, 1 ve 2) ve 0 sayfaya hemen erişildiğini, 1'e biraz daha sonra erişildiğini ve 2'e ileride en uzak sayfaya erişildiğini görüyoruz. Bu nedenle, optimum politikanın kolay bir seçeneği vardır: 2. Sayfayı tahliye edin, bu da

Erişim	Hit/Miss?	Tahliye edilcek	Ortaya çıkan Önbellek Durumu
0	Kaçırır		0
1	Kaçırır		0, 1
2	Kaçırır		0,1,2
0	Vurulduğu		0,1,2
1	Vurulduğu		0,1,2
3	Kaçırır	2	0,1,3
0	Vurulduğu		0,1,3
3	Vurulduğu		0,1,3
1	Vurulduğu		0,1,3
2	Kaçırır	3	0,1,2
1	Vurulduğu		0,1,2

Şekil 22.1: **Optimum Politikayı İzleme**

DIŞINDA: ÖNBELLEK TÜRLERI HATALI

Mimarlar bazen bilgisayar mimarisi dünyasında, yazım hatalarının karakterize edilmesinde yarar bulurlar: Zorunlu, kapasite ve çatışma kaçırmaları, bazen **de üç C'nin** [H87] adı verilen. Önbelleğin başlaması için boş olması ve bu öğenin ilk referansı olması nedeniyle zorunlu bir eksiklik (veya soğuk başlatma atlamaması [EF78]) meydana gelir; buna karşılık**, önbellekte** alan kalmadığından ve önbelleğe yeni bir öğe getirmek için bir öğeyi tahliye etmek zorunda kalmasından dolayı kapasite eksiktir. Üçüncü eksik (**bir Çakışma Güzeli**) türü, **bir öğenin bir donanım önbelleğine yerleştirilebildiği sınırlar nedeniyle donanımda ortaya çıkar;** Bu önbellekler her zaman **tamamen ilişkili olduğu için işletim sistemi sayfa önbelleğinde ortaya çıkmaz**; yani, bellekte bir sayfanın nereye yerleştirileceği konusunda herhangi bir kısıtlama yoktur. Ayrıntılar için bkz. H&P [HP06].

önbellekteki 0, 1 ve 3. sayfalar. Sonraki üç referans çok fazla. Ancak daha sonra uzun zaman önce tahliye edip bir başka eksiklik yaşadığımız 2. Sayfaya ulaşıyoruz. Burada en iyi ilke, önbellekteki her sayfa için (0, 1 ve 3) geleceği yeniden inceler ve sayfa 1'i (erişilmek üzere) göstermediği sürece sorun yaşamayacağımızı görür. Örnek, 3. Sayfanın tahliye edilmekte olduğunu, ancak 0. Sayfanın da iyi bir seçim olacağını göstermektedir. Son olarak, sayfa 1'e geldik ve izleme işlemi tamamlandı.

Ayrıca önbellek için en yüksek hızı da hesaplayabiliriz: 6 darbe ve 5 hata, vuruş oranı $\frac{\text{VURUŞLARA}}{\text{HIT'LER + MISSES}}$ bu $\frac{6}{6+5}$ veya %54.5. Ayrıca bilgisayar hit rate *modulo* zorunlu olarak hatalı (örneğin, belirli bir sayfanın ilk kaçırmasını göz ardı etme) ve %85.7 isabet oranına neden oluyor.

Ne yazık ki, planlama politikalarının geliştirilmesinde daha önce de gördüğümüz gibi, gelecek genel olarak bilinmemektedir; genel amaçlı bir işletim sistemi1 için en uygun politikayı oluşturamazsınız. Bu nedenle, gerçek ve uygulanabilir bir politika geliştirirken hangi sayfanın tahliye edileceğine karar vermenin başka bir yolunu bulacak yaklaşımlara odaklanacağız. Böylece, en iyi politika yalnızca bir karşılaştırma noktası olarak, "mükemmel" olmak için ne kadar yakın olduğumuzu bilmek için hizmet verir.

22.3 Basit bir Politika: FIFO

Çoğu eski sistem, optimum düzeyde yaklaşmaya çalışmanın karmaşıklığından kaçınmış ve çok basit değiştirme politikaları uygulamış. Örneğin, bazı sistemlerde **FIFO** (ilk giren ilk çıkar) yerine geçildi ve sayfalar sisteme girdiklerinde kuyruğa alındı; bir değiştirme işlemi gerçekleştiğinde, kuyruğun kuyruğundaki sayfa ("ilk giriş" sayfası) tahliye olur. FIFO'nun tek bir gücü vardır: Uygulanması oldukça kolaydır.

Örnek referans akışımızda FIFO'nun nasıl olduğunu inceleyelim (Şekil 22.2, sayfa 5). İzlemeye yine üç zorunlu hata ile başlıyoruz

Erişim	Hit/Miss?	Tahliye edilcek	Ortaya çıkan Önbellek Durumu	
0	Kaçırır		İlk giren →	0
1	Kaçırır		İlk giren →	0, 1
2	Kaçırır		İlk giren →	0,1,2
0	Vurulduğu		İlk giren →	0,1,2
1	Vurulduğu		İlk giren →	0,1,2
3	Kaçırır	0	İlk giren →	1,2,3
0	Kaçırır	1	İlk giren →	2,3,0
3	Vurulduğu		İlk giren →	2,3,0
1	Kaçırır	2	İlk giren →	3,0,1
2	Kaçırır	3	İlk giren →	0,1,2
1	Vurulduğu		İlk giren →	0,1,2

Şekil 22.2: **FIFO Politikasını İzleme**

sayfa 0, 1 ve 2'e ve ardından 0 ve 1'e basın. Ardından, sayfa 3'e başvurulur ve bu da bir eksiklik olmasına neden olur; FIFO ile değişim kararı kolaydır: İçindeki "ilk" sayfa olan sayfayı seçin (şekildeki önbellek durumu, ilk giriş sayfası solda olacak şekilde FIFO siparişinde tutulur), sayfa 0. Ne yazık ki, bir sonraki erişimimiz sayfa 0'e geçmektir, bu da bir başka eksiklik ve değiştirmeye neden olur (sayfa 1'in). Ardından 3. Sayfaya geldik, ancak 1. Ve 2. Sayfayı özledik ve sonunda 1. Sayfayı ziyaret ettik.

FIFO'yu optimum ile kıyasladığınızda FIFO'nun çok daha kötü bir şey olduğu ortaya çıkmıyor: %36.4'lik bir vuruş oranı (veya zorunlu atlamalar hariç %57.1). FIFO, blokların önemini belirleyemez: 0. Sayfaya birkaç kez erişilmesine rağmen, FIFO yine de bunu ortaya koymaya devam ediyor, çünkü sadece belleğe ilk katıydı.

YAN: BELADY'NİN ANORMALLİĞİ

Belady (en iyi politikanın) ve iş arkadaşları beklenmedik şekilde biraz nefret eden ilginç bir referans akışı buldular [BNS69]. Bellek referans akışı: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. Okudıkları değişim politikası FIFO idi. İlginç bölüm: 3'den 4 sayfaya kadar önbellek boyutu taşınırken önbellek vurma hızının nasıl değiştiği.

Genel olarak, önbellek büyüdüğünde önbellek vurma hızının artmasını (daha iyi olmasını) beklersiniz. Ancak bu durumda FIFO ile daha da kötüleşiyor! Vuruşları hesaplayın ve kendinizi kaçırıp görün. Bu garip davranış genellikle **Belady'nin anormalliği olarak adlandırılır** (aynı yazarların karamından).

LRU gibi diğer bazı ilkeler bu sorundan zarar görmez. Nedenini tahmin edebilir misiniz? Ortaya çıktıkça LRU **yığın özelliği** olarak bilinen [M+70] özelliğine sahiptir. Bu özelliğe sahip algoritmalar için N+1 boyutuna sahip bir önbellek doğal olarak N boyutuna sahip bir önbelleğin içeriğini içerir. Bu nedenle, önbellek boyutu artırıldığında, hit hızı aynı kalır veya artar. FIFO ve Rastgele (diğerlerinin yanı sıra) yığın özelliğine kesinlikle uymuyor ve bu nedenle anormal davranışlara karşı hassastır.

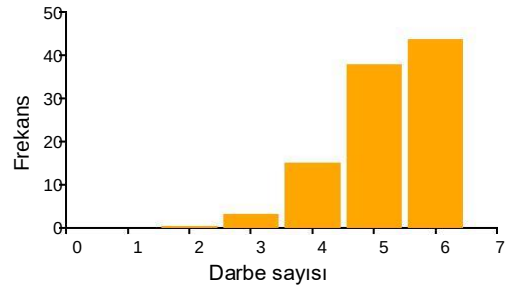
Erişim	Hit/Miss?	Tahliye edilcek	Ortaya çıkan Önbellek Durumu
0	Kaçırır		0
1	Kaçırır		0, 1
2	Kaçırır		0,1,2
0	Vurulduğu		0,1,2
1	Vurulduğu		0,1,2
3	Kaçırır	0	1,2,3
0	Kaçırır	1	2,3,0
3	Vurulduğu		2,3,0
1	Kaçırır	3	2,0,1
2	Vurulduğu		2,0,1
1	Vurulduğu		2,0,1

Şekil 22.3: **Rastgele İzleme Politikası**

22.4 Başka bir basit Politika: Rastgele

Benzer bir değiştirme ilkesi de, bellek basıncı altında değiştirilecek rastgele bir sayfa seçen Rastgele'dir. Rastgele FIFO'ya benzer özelliklere sahiptir; uygulaması kolaydır ancak hangi blokların tahliye edileceği gerçekten çok akıllı olmaya çalışmayacak. Ünlü referans akışımızda Rastgele'nin nasıl olduğuna bakalım (bkz. Şekil 22.3).

Elbette Rastgele'nin nasıl yaptığı tamamen, Rastgele'nin seçimlerine ne kadar şanslı (veya şanssız) olduğuna bağlıdır. Yukarıdaki örnekte Rastgele, FIFO'dan biraz daha iyi ve optimal'den biraz daha kötü bir şey yapar. Aslında, Rastgele denemeyi binlerce kez yürütebiliyor ve genel olarak nasıl olduğunu belirleyebiliyoruz. Şekil 22.4'de Random'da her biri farklı bir rastgele tohum içeren 10,000'den fazla denemenin kaç tanesinin elde ettiği gösterilmektedir. Gördüğünüz gibi, bazen (zamanın %40'inden biraz daha fazlası), Rastgele en iyi sonucu verir ve örnek izlemde 6 darbe alır; bazen çok daha kötü bir hal aldığından 2 veya daha az darbe elde edilir. Rastgele'nin ne yaptığı, çekilişin şansına bağlıdır.



Şekil 22.4: **10,000 denemenin üzerinde Rasgele Performans**

Erişim	Hit/Miss?	Tahliye edilcek	Ortaya çıkan Önbellek Durumu
0	Kaçırır		LRU → 0
1	Kaçırır		LRU → 0, 1
2	Kaçırır		LRU → 0,1,2
0	Vurulduğu		LRU → 1,2,0
1	Vurulduğu		LRU → 2,0,1
3	Kaçırır	2	LRU → 0,1,3
0	Vurulduğu		LRU → 1,3,0
3	Vurulduğu		LRU → 1,0,3
1	Vurulduğu		LRU → 0,3,1
2	Kaçırır	0	LRU → 3,1,2
1	Vurulduğu		LRU → 3,2,1

Şekil 22.5: **LRU Politikasını İzleme**

22.5 Geçmişi Kullanma:LRU

Ne yazık ki, FIFO veya Rastgele kadar basit bir politika yaygın bir soruna neden olabilir: Önemli bir sayfayı yeniden referans vermek üzere olabilir. FIFO ilk gelen sayfayı açar; bu, üzerinde önemli kod veya veri yapılarının bulunduğu bir sayfa olursa, kısa bir süre içinde geri çağrılsa bile her şekilde dışarı atılır. Bu nedenle, FIFO, Rastgele ve benzer politikaların optimum yaklaşma olasılığı yoktur; daha akıllı bir şey gerekiyor.

Planlama politikamız konusunda yaptığımız gibi, gelecekteki tahminimizi iyileştirmek için geçmişe bir kez daha güveniyoruz ve *geçmişimizi* kılavuz olarak kullanıyoruz. Örneğin, bir program yakın geçmişteki bir sayfaya erişmişse yakın gelecekte bu sayfaya tekrar erişim sağlayabilir.

Bir sayfa değiştirme politikasının kullanabileceği geçmiş bilgi türlerinden biri **sıklıktır**; bir sayfaya birçok kez erişilmişse, bir miktar değeri olduğu için sayfa değiştirilmemelidir. Bir sayfanın daha sık kullanılan bir özelliği, erişim yeterlilisidir; bir sayfaya ne kadar yakın zamanda erişilse, bu sayfaya yeniden erişme olasılığı da o kadar yüksek olur.

Bu politika ailesi, temelde programlar ve davranışlarıyla ilgili bir gözlem olan yerel [D70] ilkesi olarak insanların nelere atıfta bulunmalarına dayanır. Bu prensip kısaca, programların belirli kod sekanslarına (örn. Döngü içinde) ve veri yapılarına (örn. Döngü tarafından erişilen bir dizi) sıklıkla erişme eğiliminde olduğudur; bu nedenle hangi sayfaların önemli olduğunu anlamak için geçmişi kullanmaya ve tahliye zamanı geldiğinde bu sayfaları bellekte tutmaya çalışmalıyız.

Bu nedenle, tarihsel olarak basit algoritmalar ailesi doğmuştur. **En az Sık kullanılan (LFU**) politikası, bir tahliye işlemi

yapılması gereken en az sık kullanılan sayfanın yerini alır. Benzer şekilde**, en son kullanılan (LRU**) ilkesi en az kullanılan sayfanın yerini alır. Bu algoritmaları hatırlamak kolaydır: Adı öğrendikten sonra ne yaptığını tam olarak bilirsiniz, bu da bir isim için mükemmel bir özellik.

LRU'yu daha iyi anlamak için LRU'nun sınavımızda nasıl çalıştığını inceleyelim –

YAN: LOKALITE TÜRLERI

Programların sergileme eğiliminde olduğu iki lokalite türü vardır. Birincisi **uzamsal lokalite olarak bilinir** ve bu, P sayfasına erişildiğinde bunun etrafındaki sayfalara da (P − 1 veya P + 1 şeklinde) erişilebileceğini belirtir. İkincisi**,** yakın geçmişte erişilen sayfalara yakın gelecekte tekrar erişilebileceğini belirten geçici yerellik. Bu tür yersellik varlığının varsayımı, donanım sistemlerinin önbelleğe alma hiyerarşisinde büyük bir rol oynar ve bu tür bir mevki söz konusu olduğunda programların hızlı çalışmasına yardımcı olmak için çok sayıda yönerge, veri ve adres çevirisi önbelleğe alma düzeyi kullanılır.

Tabii **ki, yer tespiti ilkesi**, genellikle çağrıldıkça, tüm programların uyması gereken zor ve hızlı bir kural değildir. Gerçekten de bazı programlar belleğe (veya diske) rastgele bir şekilde erişir ve erişim akışlarında çok fazla yer veya yer göstermez. Bu nedenle, yerellik her türlü önbellekleri (donanım veya yazılım) tasarlarken akılda tutmanız gereken iyi bir şey olsa da, başarıyı garanti etmez. Bunun yerine, genellikle bilgisayar sistemlerinin tasarımında yararlı olduğunu kanıtlayan bir bulgusal özelliktir.

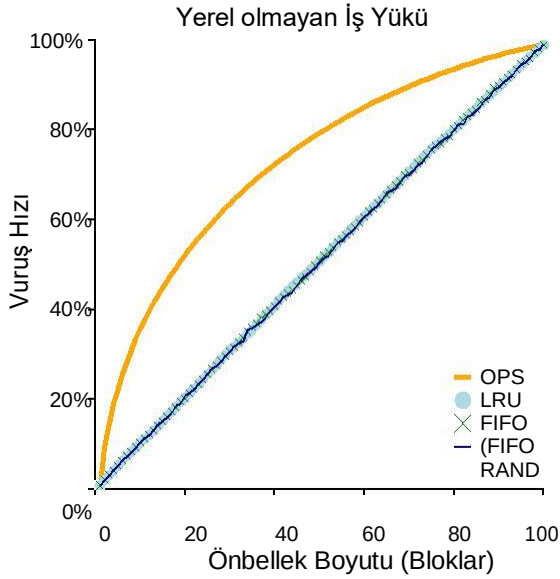
ple referans akışı. Şekil 22.5 (sayfa 7) sonuçları gösterir. Şekilden, LRU'nun Rastgele veya FIFO gibi durumsuz politikalardan daha iyisini yapmak için geçmişi nasıl kullanabileceğini görebilirsiniz. Örnekte, LRU, 0 ve 1'e daha yakın bir tarihte erişildiği için sayfa 2'i bir sayfayı değiştirmeniz gerektiğinde tahliye eder. Daha sonra 0. Sayfanın yerini alır, çünkü 1 ve 3 numaralı sayfalara daha yakın zamanda erişildi. Her iki durumda da LRU'nun kararı tarihe göre doğru çıkar ve bu nedenle sonraki referanslar isabetli olur. Bu nedenle örneğimizde LRU mümkün olduğu kadar iyi performans gösterir2.

Bu algoritmaların zıt anlamlılarının da mevcut olduğunu da unutmamalıyız: **En Sık kullanılan (MFU**) ve **en son kullanılan (MRU**). Çoğu durumda (tümü değil!) bu politikalar işe yaramaz, çünkü çoğu programın sergilendiği yerelliği benimsemek yerine görmezden gelirler.

22.6 İş yükü örnekleri

Bu politikalardan bazılarının nasıl davrandığını daha iyi anlamak için birkaç örneğe daha bakalım. Burada, küçük izlemeler yerine daha karmaşık iş yüklerini inceleyeceğiz. Ancak bu iş yükleri bile büyük ölçüde basitleştirilmiştir; daha iyi bir çalışma, uygulama izlerini içerir.

İlk iş yükümüzün yerelliği yoktur, yani her referans, erişilen sayfalar kümesi içinde rastgele bir sayfa anlamına gelir. Bu basit örnekte, iş yükü zaman içinde 100 benzersiz sayfaya erişir ve rastgele referans almak için bir sonraki sayfayı seçin; genel olarak, 10,000 sayfaya erişilir. Deneyde, her ilkenin çeşitli önbellek boyutları üzerinde nasıl davrandığını görmek için önbellek boyutunu çok küçük (1 sayfa) boyutundan tüm benzersiz sayfaları (100 sayfa) tutmaya kadar değiştiyoruz.



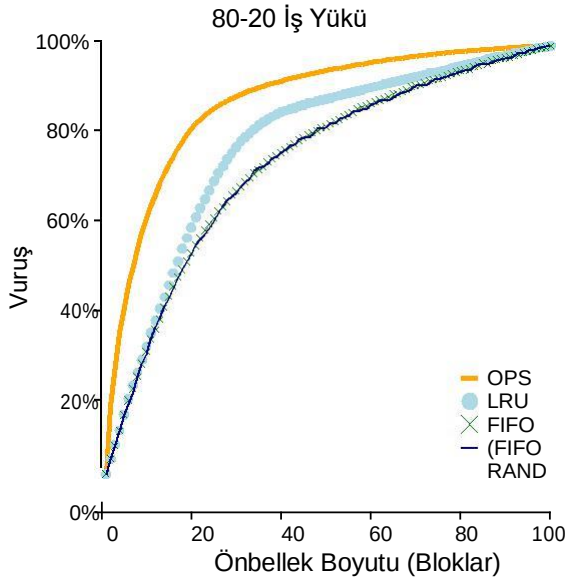
Şekil 22.6: **Yerel olmayan İş Yükü**

Şekil 22.6 en İyi, LRU, Rastgele ve FIFO deneyinin sonuçlarını çizer. Rakamın y ekseni, her ilkenin elde ettiği darbe oranını gösterir; x ekseni yukarıda açıklandığı gibi önbellek boyutunu değiştirir.

Grafikten birkaç sonuç elde edebiliriz. Öncelikle, iş yükünde bir yer olmadığında, hangi gerçekçi politikayı kullandığınız önemli değildir; LRU, FIFO ve Rastgele de aynı performansı gösterir ve vuruş hızı önbelleğin boyutuna göre tam olarak belirlenir. İkinci olarak, önbellek tüm iş yüküne sığacak kadar büyük olduğunda, hangi ilkeyi kullandığınız da önemli değildir; Tüm ilkeler (Rastgele bile), referans verilen tüm bloklar önbelleğe sığdığında %100'lik bir darbe oranına yakınıyor. Son olarak, optimum performans performansının gerçekçi politikalardan belirgin şekilde daha iyi olduğunu görebilirsiniz; gelecekte mümkün olduğu takdirde, değişim için çok daha iyi bir iş çıkarıyorlar.

İncelediğimiz bir sonraki iş yüküne "80-20" iş yükü denir ve bu iş yükü yerelliği gösterir: Referansların %80'i sayfaların %20'ine ("sıcak" sayfalar) yapılır; referansların kalan %20'i sayfaların kalan %80'ine ("soğuk" sayfalar) yapılır. İş yükümüzde, yine toplam 100 benzersiz sayfa vardır; bu nedenle, "sıcak" sayfalar çoğu zaman, "soğuk" sayfalar ise geri kalan sayfalardır. Şekil 22.7 (sayfa 10) politikaların bu iş yüküyle nasıl performans gösterdiğini gösterir.

Hem rastgele hem de FIFO makul ölçüde iyi sonuç verse de, LRU daha iyi iş görür, çünkü sıcak sayfaların üzerinde tutunması daha olasıdır; bu sayfalara geçmişte sık sık başvurulan sayfalar, yakın gelecekte yeniden başvurma olasılığındadır. Bir kez daha optimum hale getirmek, LRU'nun geçmiş bilgilerinin mükemmel olmadığını gösterir.

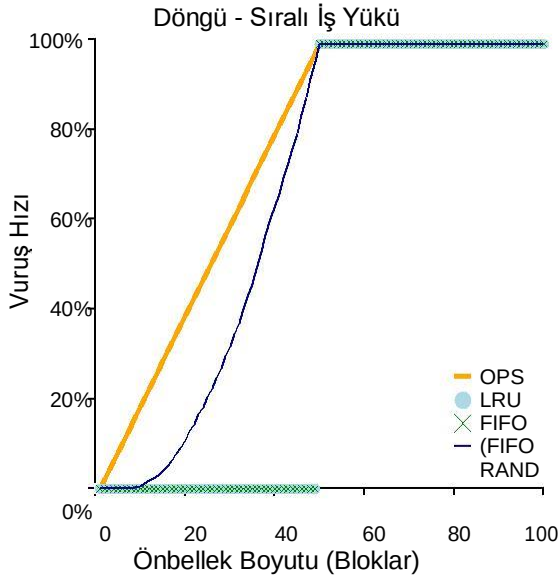


Şekil 22.7: **80-20 İş Yükü**

Şimdi merak ediyor olabilirsiniz: LRU'nun Rastgele ve FIFO üzerindeki gelişimi gerçekten bu kadar büyük mü? "Her zamanki gibi, yanıt "değişir." Her bir eksiklik çok pahalıya mal olursa (sık rastlanılmayan bir durum olmasa da), yüksek fiyat artışı (kaçırma oranındaki düşüş) bile performansta büyük bir fark yaratabilir. Atlayışlar çok maliyetli değilse tabii ki LRU ile mümkün olan avantajlar da neredeyse hiç önemli değildir.

Şimdi son bir iş yüküne bakalım. Buna "döngüsel sıralı" iş yükü diyoruz, bu iş yükünde olduğu gibi, sırasıyla 0, 1,..., sayfa 49'den başlayarak 50 sayfaya kadar, daha sonra bu erişimleri tekrarlayarak 50 benzersiz sayfaya toplam 10,000 erişim için döngü yapıyoruz. Şekil 22.8'deki son grafik, bu iş yükü altındaki ilkelerin davranışını gösterir.

Birçok uygulamada ortak olan bu iş yükü (veritabanları [CD85] gibi önemli ticari uygulamalar dahil), hem LRU hem de FIFO için en kötü durumu temsil eder. Döngüsel sıralı iş yükü altında bulunan bu algoritmalar, eski sayfaları ortaya koyar; ne yazık ki, iş yükünün döngü yapısı nedeniyle bu eski sayfalara, ilkelerin önbellekte tutmayı tercih ettiği sayfalardan daha çabuk erişilebilecek. Gerçekten de, 49 boyutlu bir önbellekle bile, 50 sayfalık döngüsel sıralı iş yükü %0'lik bir darbe oranına neden olur. İlginç bir şekilde, Rastgele belirgin şekilde daha iyi bir şekilde ilerliyor, en azından sıfır olmayan vuruş oranına ulaşıyor. Rastgele gelişin bazı güzel özellikleri olduğunu ortaya çıkarmıştır; bu tür bir işletme tuhaf köşe durum davranışlarına sahip değil.



Şekil 22.8: **Döngü İş Yükü**

22.7 Geçmiş algoritmaları Uygulama

Gördüğünüz gibi, LRU gibi bir algoritma genellikle FIFO veya Rastgele gibi daha basit ilkelerden daha iyi bir iş yapabilir ve bu da önemli sayfaları yok edebilir. Ne yazık ki tarihi politikalar bizi yeni bir zorlukla tanıştırıyor: Bunları nasıl uygularız?

Örneğin LRU'yu ele alalım. Mükemmel bir şekilde uygulamak için çok fazla iş yapmamız gerekir. Özellikle, *her sayfa erişiminden sonra* (örn. Her bellek erişiminde, talimat alma veya yükleme ya da depolama), bu sayfayı listenin önüne (örn. MRU tarafı) taşımak için bazı veri yapısını güncellememiz gerekir. Bunu, FIFO sayfa listesine yalnızca bir sayfa *çıkarıldığında* (ilk giriş sayfası kaldırılarak) veya listeye yeni bir sayfa eklendiğinde (son giriş tarafına) erişilebildiği FIFO ile karşılaştırın. Hangi sayfaların en az ve en son kullanıldığını takip etmek için sistemin *her bellek referansı üzerinde bazı hesaplama çalışmaları yapması gerekir.* Net bir şekilde, büyük özen göstermeden, bu tür muhasebe performansı büyük ölçüde düşürebilir.

Bunu hızlandırmaya yardımcı olabilecek yöntemlerden biri, biraz donanım desteği ekmektir. Örneğin, bir makine her sayfa erişiminde bellekteki bir saat alanını güncelleyebilir (örneğin, bu işlem başına sayfa tablosunda veya yalnızca fiziksel sayfa başına bir girişle bellekteki bazı ayrı dizide olabilir). Bu nedenle, bir sayfaya erişildiğinde, saat alanı donanım tarafından geçerli zamana ayarlanır. Ardından, bir sayfayı değiştirirken, en son kullanılan sayfayı bulmak için işletim sistemi sistemdeki tüm zaman alanlarını tarayabilir.

Ne yazık ki, bir sistemdeki sayfa sayısı arttıkça, en son kullanılan mutlak sayfayı bulmak için çok sayıda tarama yapmak oldukça pahalıdır. 4 GB belleğe sahip, 4 KB sayfalara doğranmış modern bir makine hayal edin. Bu makine 1 milyon sayfaya sahiptir ve böylece modern CPU hızlarında bile LRU sayfasının bulunması uzun sürer. Hangi soru soruluyor: Değiştirmek için en eski mutlak sayfayı gerçekten bulmamız gerekiyor mu? Bunun yerine yaklaşık olarak hayatta kalabilecek miyiz?

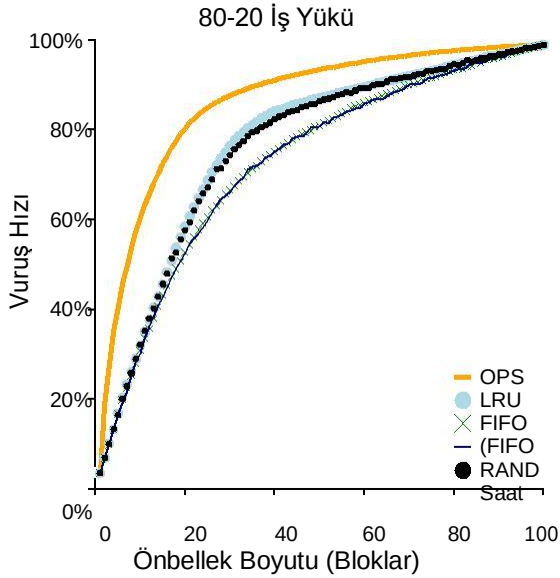
HAÇLI: Mükemmel LRU'yu uygulamak pahalıya mal olacağı için LRU DEĞİŞTIRME POLITIKASINI NASIL UYGULAYABILIRIZ, ap-
bunu bir şekilde yapın ve yine de istenen davranışı elde edin.

22.8 Yaklaşık LRU

Sonuç olarak yanıt evet: LRU'nun hesaplanması, bilgisayar genel bakış açısından daha uygulanabilir ve aslında birçok modern sistemin yaptığı şeydir. Fikir, ilk olarak sayfalama ile ilk sistemde uygulanan bir kullanım biti (bazen referans biti olarak da adlandırılır) biçiminde, Atlas tek düzeyli mağazası [KE+62] olan bir donanım desteği gerektirir. Sistemin her sayfası için bir kullanım biti vardır ve kullanım bitleri bellekte bir yerde bulunur (bunlar süreç başına sayfa tablolarında, örneğin bir dizide veya bir yerde olabilir). Bir sayfaya her başvuruda bulunduğumda (yani, okuma veya yazma), kullanım biti donanım tarafından 1 olarak ayarlanır. Donanım hiçbir zaman bit'i temizlemese de (yani, 0 olarak ayarlar); Bu, işletim sisteminin sorumluluğudur.

İşletim sisteminde yaklaşık LRU için Kullanım biti nasıl kullanılır? Birçok yol olabilir, ancak **saat algoritması** [C69] ile basit bir yaklaşım önerildi. Sistemin tüm sayfalarının dairesel bir liste halinde düzenlendiğini düşünün. **Bir saat belirli** bir sayfaya işaret eder (hangisi önemli değildir). Değiştirilmesi gerektiğinde, işletim sistemi o anda P sayfasında 1 veya 0 bit kullanımı olup olmadığını kontrol eder. 1 ise bu, P sayfasının yakın zamanda kullanıldığını ve dolayısıyla değiştirilmesi için iyi bir aday olmadığını ima eder. Bu nedenle, P için kullanım biti 0 olarak ayarlanır (temizlenir) ve saat eli sonraki sayfaya (P+1) artırılır. Algoritma, 0 olarak ayarlanmış bir kullanım biti bulana kadar devam eder ve bu, bu sayfanın yakın zamanda kullanılmadığını (veya en kötü durumda tüm sayfaların kullanıldığını ve artık tüm sayfa kümesinde arama yaptığımızı, tüm bitleri temizlediğimizi) ima eder.

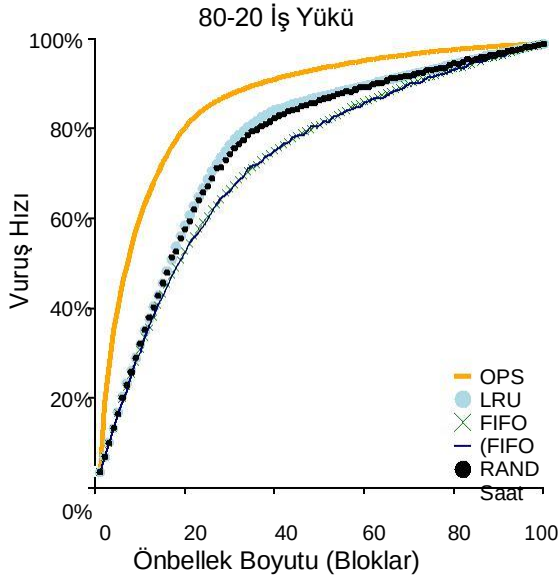
Bu yaklaşımın yaklaşık LRU'da kullanım biti kullanmanın tek yolu olmadığını unutmayın. Gerçekten de, kullanım bitlerini düzenli olarak temizleyen ve hangi sayfaların hangi sayfaların hangi sayfaların hangi sayfaların değiştirileceğine karar vermek için 1 ile 0 arasında bit kullanımı olduğunu ayıran tüm yaklaşımlar iyi olacaktır. Corbato'nun saat algoritması, bir miktar başarıyla karşılaşan sadece bir erken yaklaşım ve kullanılmayan bir sayfa arayan tüm bellekleri sürekli olarak tarayamayan güzel bir özelliğe sahipti.



Şekil 22.9: **Saatli 80-20 İş Yükü**

Saat algoritması varyantı Şekil 22.9'de gösterilmiştir. Bu isteğe bağlı donanım, bir değiştirme işlemi sırasında sayfaları rastgele tarar; referans biti 1 olarak ayarlanmış bir sayfayla karşılaştığında, biti temizler (yani, 0 olarak ayarlar); referans biti 0 olarak ayarlanmış bir sayfa bulduğunda, kurban olarak bu sayfayı seçer. Gördüğünüz gibi, mükemmel LRU kadar iyi olmasa da, geçmişi dikkate almayan yaklaşımlardan daha iyidir.

,



Şekil 22.9: **Saatli 80-20 İş Yükü**

Saat algoritması varyantı Şekil 22.9'de gösterilmiştir. Bu isteğe bağlı donanım, bir değiştirme işlemi sırasında sayfaları rastgele tarar; referans biti 1 olarak ayarlanmış bir sayfayla karşılaştığında, biti temizler (yani, 0 olarak ayarlar); referans biti 0 olarak ayarlanmış bir sayfa bulduğunda, kurban olarak bu sayfayı seçer. Gördüğünüz gibi, mükemmel LRU kadar iyi olmasa da, geçmişi dikkate almayan yaklaşımlardan daha iyidir.

22.9 Kirli sayfalar göz önünde bulundurulduğunda

Saat algoritmasında yapılan küçük bir değişiklik (başlangıçta Corbato [C69] tarafından da önerilir), yaygın olarak yapılan bir değişikliktir ve bir sayfanın bellekteyken değiştirilip değiştirilmediği ile ilgili ek bir değerlendirmedir. Bunun nedeni: Bir sayfa **değiştirilmiş** ve bu nedenle kirliyse, sayfayı çıkarmak için diske geri yazılmalıdır, bu pahalı bir şeydir. Değişiklik yapılmazsa (ve bu nedenle temizse) tahliye ücretsizdir; Fiziksel çerçeve, ek G/Ç gerektirmeden başka amaçlar için yeniden kullanılabilir Bu nedenle, bazı VM sistemleri kirli sayfaların üzerine temiz sayfaları boşaltmayı tercih eder.

Bu davranışı desteklemek için donanım **değiştirilmiş bir bit (**yani **kirli bir bit**) içermelidir. Bu bit, bir sayfa her yazıldığında ayarlanır ve bu nedenle sayfa değiştirme algoritmasına eklenebilir. Örneğin saat algoritması, hem kullanılmayan hem de temiz olan sayfaları tarayarak ilk önce tahliye etmek üzere değiştirilebilir; bunları bulamamak, kirli olan kullanılmayan sayfaları bulmak vb.

22.10 Diğer VM İlkeleri

Sayfa değiştirme, VM alt sisteminin kullandığı tek ilke değildir (ancak en önemli ilke olabilir). Örneğin, işletim sisteminin sayfanın belleğe ne zaman getirileceğine de karar vermesi gerekir. **Bazen sayfa seçme** ilkesi adı verilen bu ilke (Denning [D70] (Reddetme [3]) ile çağrıldıkça) işletim sistemini farklı seçenekler sunar.

İşletim sistemi çoğu sayfa için **talep sayfalama özelliğini kullanır**, yani işletim sistemi, erişildiğinde sayfayı belleğe, "talep üzerine" olduğu gibi getirir. Elbette, işletim sistemi bir sayfanın kullanılmak üzere olduğunu tahmin edebilir ve bu nedenle sayfayı önceden getirebilir; bu davranış **ön belleğe alma olarak bilinir** ve yalnızca makul bir başarı ihtimali olduğunda yapılır. Örneğin, bazı sistemlerde kod sayfası P belleğe getirilirse o kod sayfasına P +1 yakında erişilebileceği ve dolayısıyla belleğe de getirilebileceği varsayılır.

Başka bir ilke, işletim sisteminin sayfaları diske nasıl yazacağını belirler. Tabi ki, tek tek yazılabilirler; ancak, birçok sistem bunun yerine bellekte bir dizi bekleyen yazıyı toplar ve bunları tek (daha verimli) yazıda diske yazar. Bu davranış genellikle **kümeleme** veya yazma gruplama olarak adlandırılır ve tek bir büyük yazımı birçok küçük sürücüden daha verimli bir şekilde gerçekleştiren disk sürücülerinin yapısı nedeniyle etkilidir.

22.11 Vurma

Kapatmadan önce bir son soruya değineceğiz: Bellek yalnızca aşırı abone olduğunda ve çalışan işlemlerin bellek talepleri kullanılabilir fiziksel belleği aştığında işletim sisteminin ne yapması gerekir? Bu durumda sistem sürekli olarak çağrı yapar ve bazen **tahkil** [D70] olarak adlandırılır.

Önceki bazı işletim sistemlerinde, gerçekleştiğinde tahkili tespit etmek ve bu tahkiyle başa çıkmak için oldukça gelişmiş bir mekanizma seti vardı. Örneğin, bir dizi işlem göz önüne alındığında, bir sistem bir süreç alt kümesini çalıştırmamaya karar verebilir ve azaltılmış işlem kümesinin (etkin olarak kullandıkları sayfalar) belleğe sığması ve böylece ilerleme kaydedebilmesi umut eder. Genel olarak **kabul denetimi olarak bilinen bu yaklaşım**, her şeyi bir kerede kötü bir şekilde yapmaya çalışmak yerine bazen daha az işi daha iyi bir şekilde yapmanın daha iyi olduğunu, sıklıkla gerçek hayatta ve modern bilgisayar sistemlerinde (ne yazık ki) karşılaştığımız bir durumu belirtir.

Bazı mevcut sistemler, aşırı bellek yükü için daha draconian yaklaşımını benimsiyor. Örneğin, Linux'un bazı sürümlerinde bellek aşırı abone olduğunda bellek yetersiz katil çalışır; bu daemon bellek yoğun bir işlemi seçer ve bunu öldürür, böylece bellek çok ince bir şekilde azaltılır. Bellek basıncını azaltmada başarılı olsa da, örneğin X sunucusunu öldürüp ekranı kullanılmaz hale getirdiği takdirde, bu yaklaşımda sorunlar olabilir.

22.12 Özet

Tüm modern işletim sistemlerinin VM alt sisteminin bir parçası olan bir dizi sayfa değiştirme (ve diğer) politikasını kullanıma sunduk. Modern sistemler, saat gibi basit LRU yaklaşımlarına bazı ince ayarlar ekler; Örneğin, **tarama direnci** ARC [MM03] gibi birçok modern algoritmanın önemli bir parçasıdır. Tarama direnci algoritmaları genellikle LRU benzeri olmakla birlikte döngüsel sıralı iş yüküyle gördüğümüz LRU'nun en kötü durum davranışını önlemeye de çalışır. Böylece, sayfa değiştirme algoritmalarının gelişimi devam eder.

Bellek erişimi ve disk erişim süreleri arasındaki farklılık çok büyük olduğundan, uzun yıllar boyunca değiştirme algoritmalarının önemi azalmıştır. Özellikle, diske gitmek çok pahalı olduğu için, sık sık çağrı yapmak pahalıydı; tek yapmanız gereken, değiştirme algoritmanız ne kadar iyi olursa olsun, sık sık değiştirme yapıyorsan sisteminiz inanılmaz derecede yavaşladı. Bu nedenle, en iyi çözüm basit (zekice tatmin edici değilse) bir çözümdü: Daha fazla bellek satın alın.

Ancak, çok daha hızlı depolama aygıtlarında (örn. Flash tabanlı SSD'ler ve Intel Optane ürünleri) son zamanlarda yapılan yenilikler, bu performans oranlarını bir kez daha değiştirmiş ve sayfa değiştirme algorasyonlarında yeniden bir rönesans oluşturmuştur. Bu alandaki son çalışmalar için bkz. [SS10, W+21].

References

[AD03] "Run-Time Adaptation in River" by Remzi H. Arpaci-Dusseau. ACM TOCS, 21:1, February 2003. *A summary of one of the authors' dissertation work on a system named River, where he learned that comparison against the ideal is an important technique for system designers.*

[B66] "A Study of Replacement Algorithms for Virtual-Storage Computer" by Laszlo A. Be- lady. IBM Systems Journal 5(2): 78-101, 1966. *The paper that introduces the simple way to compute the optimal behavior of a policy (the MIN algorithm).*

[BNS69] "An Anomaly in Space-time Characteristics of Certain Programs Running in a Paging Machine" by L. A. Belady, R. A. Nelson, G. S. Shedler. Communications of the ACM, 12:6, June 1969. *Introduction of the little sequence of memory references known as Belady's Anomaly. How doNelson and Shedler feel about this name, we wonder?*

[CD85] "An Evaluation of Buffer Management Strategies for Relational Database Systems" by Hong-Tai Chou, David J. DeWitt. VLDB '85, Stockholm, Sweden, August 1985. *A famous database paper on the different buffering strategies you should use under a number of common database access patterns. The more general lesson: if you know something about a workload, you can tailor policies to do better than the general-purpose ones usually found in the OS.*

[C69] "A Paging Experiment with the Multics System" by F.J. Corbato. Included in a Festschrift published in honor of Prof. P.M. Morse. MIT Press, Cambridge, MA, 1969. *The original (and hard to find!) reference to the clock algorithm, though not the first usage of a use bit. Thanks to H. Balakrishnan of MIT for digging up this paper for us.*

[D70] "Virtual Memory" by Peter J. Denning. Computing Surveys, Vol. 2, No. 3, September 1970. *Denning's early and famous survey on virtual memory systems.*

[EF78] "Cold-start vs. Warm-start Miss Ratios" by Malcolm C. Easton, Ronald Fagin. Commu- nications of the ACM, 21:10, October 1978. *A good discussion of cold- vs. warm-start misses.*

[FP89] "Electrochemically Induced Nuclear Fusion of Deuterium" by Martin Fleischmann, Stanley Pons. Journal of Electroanalytical Chemistry, Volume 26, Number 2, Part 1, April, 1989. *The famous paper that would have revolutionized the world in providing an easy way to generate nearly-infinite power from jars of water with a little metal in them. Unfortunately, the results pub- lished (and widely publicized) by Pons and Fleischmann were impossible to reproduce, and thus these two well-meaning scientists were discredited (and certainly, mocked). The only guy really happy about this result was Marvin Hawkins, whose name was left off this paper even though he participated in the work, thus avoiding association with one of the biggest scientific goofs of the 20th century.*

[HP06] "Computer Architecture: A Quantitative Approach" by John Hennessy and David Patterson. Morgan-Kaufmann, 2006. *A marvelous book about computer architecture. Read it!*

[H87] "Aspects of Cache Memory and Instruction Buffer Performance" by Mark D. Hill. Ph.D. Dissertation, U.C. Berkeley, 1987. *Mark Hill, in his dissertation work, introduced the Three C's, which later gained wide popularity with its inclusion in H&P [HP06]. The quote from therein: "I have found it useful to partition misses ... into three components intuitively based on the cause of the misses (page 49)."*

[KE+62] "One-level Storage System" by T. Kilburn, D.B.G. Edwards, M.J. Lanigan, F.H. Sum- ner. IRE Trans. EC-11:2, 1962. *Although Atlas had a use bit, it only had a very small number of pages, and thus the scanning of the use bits in large memories was not a problem the authors solved.*

[M+70] "Evaluation Techniques for Storage Hierarchies" by R. L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger. IBM Systems Journal, Volume 9:2, 1970. *A paper that is mostly about how to simulate cache hierarchies efficiently; certainly a classic in that regard, as well for its excellent discussion of some of the properties of various replacement algorithms. Can you figure out why the stack property might be useful for simulating a lot of different-sized caches at once?*

[MM03] "ARC: A Self-Tuning, Low Overhead Replacement Cache" by Nimrod Megiddo and Dharmendra S. Modha. FAST 2003, February 2003, San Jose, California. *An excellent modern paper about replacement algorithms, which includes a new policy, ARC, that is now used in some systems. Recognized in 2014 as a "Test of Time" award winner by the storage systems community at the FAST '14 conference.*

[SS10] "FlashVM: Virtual Memory Management on Flash" by Mohit Saxena, Michael M. Swift. USENIX ATC '10, June, 2010, Boston, MA. *An early, excellent paper by our colleagues at U. Wisconsin about how to use Flash for paging. One interesting twist is how the system has to take* ***wearout****, an intrinsic property of Flash-based devices, into account. Read more about Flash-based SSDs later in this book if you are interested.*

[W+21] "The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Stor- age Devices with Orthus" by Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau. FAST '21, held virtually – thanks, COVID-19. *Our own work on a caching approach on modern devices; is it directly related to page replacement? Perhaps not. But it is a fun paper.*

Ev ödevi (Simülasyon)

`Bu simülatör, çağrı-politikası.py`, farklı sayfa değiştirme ilkeleriyle oynamanıza olanak tanır. Ayrıntılar için BENIOKU DOSYASINA bakın.

Sorular

1. Aşağıdaki bağımsız değişkenlerle rasgele adresler oluşturun: `-S 0 -n 10`, `-s 1 -n 10` ve `-s 2 -n 10`. FIFO politikasını LRU olarak değiştirerek TERCIH edin. Söylenmiş adres izlemlerinde her erişimin hit mi yoksa yanlış mı olduğunu hesaplayın.

```
PS C:\Users\Ahmet Emin> python3 ./paging-policy.py -s 0 -n 10
>> ARG addresses -1
>> ARG addressfile
>> ARG numaddrs 10
>> ARG policy FIFO
>> ARG clockbits 2
>> ARG cachesize 3
>> ARG maxpage 10
>> ARG seed 0
>> ARG notrace False
>>
>> Assuming a replacement policy of FIFO, and a cache of size 3 pages,
>> figure out whether each of the following page references hit or miss
>> in the page cache.
>>
>> Access: 8  Hit/Miss?  State of Memory?
>> Access: 7  Hit/Miss?  State of Memory?
>> Access: 4  Hit/Miss?  State of Memory?
>> Access: 2  Hit/Miss?  State of Memory?
>> Access: 5  Hit/Miss?  State of Memory?
>> Access: 4  Hit/Miss?  State of Memory?
>> Access: 7  Hit/Miss?  State of Memory?
>> Access: 3  Hit/Miss?  State of Memory?
>> Access: 4  Hit/Miss?  State of Memory?
>> Access: 5  Hit/Miss?  State of Memory?
```

FIFO İLE BERABER

Access	Cache	Outcome
8	[]	miss
7	[8]	miss
4	[8, 7]	miss
2	[8, 7, 4]	miss
5	[7, 4, 2]	miss
4	[4, 2, 5]	hit
7	[4, 2, 5]	miss
3	[2, 5, 7]	miss
4	[5, 7, 3]	miss
5	[7, 3, 4]	miss

LRU İLE BERABER

Access	Cache	Outcome
8	[]	miss
7	[8]	miss
4	[8, 7]	miss
2	[8, 7, 4]	miss
5	[7, 4, 2]	miss
4	[4, 2, 5]	hit
7	[4, 2, 5]	miss
3	[4, 5, 7]	miss
4	[4, 7, 3]	hit
5	[4, 7, 3]	miss

OPT İLE BERABER

Access	Cache	Outcome
8	[]	miss
7	[8]	miss
4	[8, 7]	miss
2	[8, 7, 4]	miss
5	[7, 4, 2]	miss
4	[7, 4, 5]	hit
7	[7, 4, 5]	hit
3	[7, 4, 5]	miss
4	[4, 5, 3]	hit
5	[4, 7, 3]	hit

2. 5 boyutundaki bir önbellek için, aşağıdaki ilkelerin her biri için en kötü durum adres referans akışları oluşturun: FIFO, LRU ve MRU (en kötü durum referans akışları, mümkün olan en kötü hatalara neden olur. En kötü durum referans akışlarında performansı önemli ölçüde artırmak ve ISTEĞE yaklaşmak için önbelleğin ne kadar daha büyük olması gerekir?

FIFO:

```
python3 ./paging-policy.py -s 0 --policy=FIFO -C 5 -a 0,1,2,3,4,5,0,1,2,3,4,5,0,1,2,3,4,5 -c
```

LRU:
Aynı giriş, LRU için de en kötü durumdur. Önbellek boyutunun 1 artırılması, her iki durumda DA AYNI sonucu verir.

MRU:

```
python3 ./paging-policy.py -s 0 --policy=MRU -C 5 -a 0,1,2,3,4,5,4,5,4,5,4,5,4,5,4,5,4,5 -c
```

3. Rastgele bir iz oluşturun (python veya perl kullanın). Bu tür bir izlemde farklı politikaların nasıl performans göstermesini beklersiniz?

LRU, MRU, FIFO rastgele sekanslarda iyi veya kötü performans gösterme şansına sahiptir. Bu sırada FIFO en iyi performansı %50 vuruş hızıyla yapar. LRU %43 ile biraz daha kötü çalışıyor. MRU %25 ile en kötü şekilde çalışıyor

BU KODU ÇALIŞTIRDIĞIMIZDA ÇIKTI AŞŞAĞIDAKİ GİBİDİR.

```
PS C:\Users\Ahmet Emin> python3 ./paging-policy.py -s 0 --policy=LRU -C 4 -a 7,5,1,5,1,7,5,3,3,0,4,1,3,5,2,5,6,0,1,2,3,0
,8,1,0,1,3,1,6,1,5,1 -c
```

4. Şimdi bazı yerelliğe sahip bir iz oluşturun. Böyle bir izi nasıl oluşturabilirsiniz? LRU'nun performansı nedir? LRU, RAND'den ne kadar daha iyidir? SAAT nasıl çalışıyor? Saat kaç farklı sayıda saat bitine sahip?

5. Gerçek bir uygulamayı enstrümanlamak ve sanal bir sayfa referans akışı oluşturmak için valgrind gibi bir program kullanın. Örneğin, çalışan `valgrind --tool=lackey --trace-mem=Yes ls` programı `ls` tarafından yapılan her talimat ve veri referansı için neredeyse eksiksiz bir referans izi sağlar. Bunu yukarıdaki simülatör için yararlı hale getirmek için öncelikle her bir sanal bellek referansını sanal bir sayfa numarası referansına dönüştürmeniz gerekir (ofseti maskeleyerek ve ortaya çıkan bitleri aşağı kaydırarak yapılır). Çok küçük istekleri karşılamak için uygulama iziniz için ne kadar büyük önbellek gerekir? Önbelleğin boyutu arttıkça çalışma kümasının bir grafiğini çizin.

Sayfalama sayesinde, 1 önbellek boyutunda bile %50 isabet sağlar.