22

Fiziksel Belleğin Ötesinde: Politikalar (Beyond Physical Memory: Policies)

Bir sanal makine yöneticisinde, çok fazla boş belleğiniz olduğunda hayat kolaydır. Bir sayfa hatası meydana gelir, boş sayfalar listesinden boş bir sayfa bulursunuz,ve hata veren sayfaya atarsınız. Hey, İşletim Sistemi, tebrikler! Yine başardın.

Maalesef, Az bellek olduğunda işler biraz daha ilginçleşiyor. Böyle bir durumda, bu **bellek baskısı (memory pressure**) etkin olarak kullanılan sayfalara yer açmak için işletim sistemini **sayfalama (paging out)** yapmaya zorlar. Hangi sayfanın (ya da sayfaların) çıkartılacağına karar verilmesi İşletim Sisteminin **değiştirme politikasında (replacement policy)** özetlenmiştir; tarihsel olarak, eski sistemlerde çok az bir fiziksel bellek bulunduğu için, erken dönem sanal belllek sistemlerinin verdiği en önemli kararlardan biriydi.Asgari düzeyde, bu hakkında biraz daha bilmeye değer olan bir ilginç bir dizi politikadır. Ve böylece problemimiz:

> Can Alıcı Nokta: Hangi Sayfanın Çıkarılacağına Nasıl Karar Verilir (THE CRUX: HOW TO DECIDE WHICH PAGE TO EVICT)
> İşletim Sistemi, bellekten hangi sayfaların çıkarılacağına nasıl karar verir? Bu karar systemin değiştirme politikası tarafından verilir ki genellikle bazı prensipleri takip eder (aşağıda tartışılmıştır) ancak aynı zamanda köşe durum davranışlarından kaçınmak için kesin ayarlamalar da içerir.

22.1 Önbellek Yönetimi (Cache Management)

Politikalara dalmadan önce, ilk olarak çözmeye çalıştığımız problemi biraz daha detaylandıralım. Ana belleğin sistemdeki tüm sayfaların bazı alt kümelerini tuttuğu göz önüne alındığında, haklı olarak sistemdeki sanal belAylek sayfaları için bir önbellek olarak görünebilir. Bu yüzden, bu önbellek için bir değiştirme politikası seçmedeki amacımız **bellek kayıplarının (cache misses)** sayısını en aza indirmektir. Yani, diskten bir sayfa getirmemiz gereken sayıyı en aza indirmektir. Alternatif olarak, hedefimiz önbellek isabetlerinin sayısını en üst düzeye çıkarmak olarak görülebilir. Yani, erişilen bir sayfanın bellekte bulunma sayısı. Önbellek isabetlerinin ve kayıplarının sayısını bilmek bir program için **ortalama bellek erişim süresini (average memory access time (AMAT))** hesaplamamızı sağlar (donanım önbellekleri için hesaplama yapan bir metrik bilgisayar mimarisi [HP06]). Özellikle, bu değerler göz önüne alındığında, bir programın ortalama bellek erişim süresini aşağıdaki gibi hesaplayabiliriz:

$$AMAT = T_M + (P_{KAYIP} \cdot T_D) \quad (22.1)$$

T_M bellek erişim maliyetini temsil eder, T_D diske erişim maliyeti, ve P_{KAYIP} önbellekte veri bulamama olasılığı (bir kayıp); P_{KAYIP} 0,0 ile 1,0 arasında değer alır ve bazen bir olasılık yerine bir yüzde kaçırma oranına atıfta bulunuruz (örneğin, %10'luk bir kaçırma oranı şu anlama gelir: $P_{KAYIP} = 0.10$). Her zaman bellekteki verilere erişmenin maliyetini ödediğinizi unutmayın; kaçırdığınız zaman, bir şekilde, ek olarak verileri diskten alma maliyetini de ödemeniz gerekir.

Örneğin, (küçük) bir adres alanına sahip bir makine hayal edelim: 256-byte sayfalarla 4KB . Böylece, bir sanal adresin iki bileşeni vardır: Bir 4-bit VPN (en anlamlı bitler) ve bir 8-bit offset (en az anlamlı bitler). Böylece, bu örnekteki bir işlem toplam 24 veya 16 sanal sayfaya erişebilir. Bu örnekte, süreç aşağıdaki bellek referanslarını meydana getirir (yani, sanal adresler): 0x000, 0x100, 0x200, 0x300, 0x400, 0x500, 0x600, 0x700, 0x800, 0x900. Bu sanal adresler, adres uzayının ilk on sayfasının her birinin ilk byte'ını ifade eder. (sayfa numarası, her sanal adresin ilk onaltılık basamağıdır).

Sanal sayfa 3 dışındaki her sayfanın zaten bellekte olduğunu varsayalım. Böylece, bellek referansları dizimiz aşağıdaki davranışla karşılaşacaktır: isabet, isabet, isabet, kayıp, isabet, isabet, isabet, isabet, isabet, isabet. **İsabet ihtimalini (hit rate)** hesaplayabiliriz (bellekte bulunan referansların yüzdesi): 90%, referansların 10'da 9'u bellekte vardır. Böylece **kayıp ihtimali (miss rate)** 10%'dur ($P_{KAYIP} = 0.1$). Genel olarak, $P_{İSABET} + P_{KAYIP} = 1.0$; isabet ihtimali ve kayıp ihtimali toplamı 100%'dür.

Ortalama bellek erişim süresini hesaplamak için, belleğe ve diske erişimin maliyetini bilmeliyiz. Belleğe erişmenin maliyetini (T_M) 100 nanosaniye civarında ve diske erişimin maliyetini de (T_D) 10 milisaniye civarında varsayarsak, Ortalama bellek erişim süresini şöyle buluruz: 100nanosaniye + 0,1 10milisaniye, yani 100nanosaniye + 1milisaniye veya 1,0001 milisaniye veya yaklaşık 1 milisaniye. Bunun yerine isabet oranımız 99.9% ($P_{KAYIP} = 0.001$) olsaydı, sonuç daha farklı olurdu: Ortalama bellek erişim süresi 10.1 mikrosaniye, ya da kabaca 100 kez daha hızlı. İsabet oranı 100%'e yaklaşırken, ortalama bellek erişim süresi 100 nanosaniyeye yaklaşır.

Maalesef, bu örnekte de göreceğiniz üzere, Modern sistemlerde disk erişiminin maliyeti o kadar yüksektir ki, küçük bir hata oranı bile çalışan programların genel ortalama bellek erişim süresine hızla hakim olacaktır. Açıkça, mümkün olduğu kadar çok kayıptan kaçınmamız veya disk hızında yavaş çalışmamız gerekiyor. Bunu çözmenin bir yolu şimdi yaptığımız gibi dikkatli bir şekilde akıllı bir politika geliştirmektir.

22.2 En Uygun Değiştirme Politikası (The Optimal Replacement Policy)

Belirli bir değiştirme politikasının nasıl çalıştığını daha iyi anlamak için, onu mümkün olan en iyi değiştirme politikasıyla karşılaştırmak iyi olur. Anlaşılacağı üzere, böyle bir en uygun politika Belady tarafından yıllar önce geliştirildi [B66] (aslen MIN olarak adlandırdı). En uygun değiştirme politikası genel olarak en az sayıda kayba yol açar. Belady, gelecekte en uzağa erişilecek sayfanın yerini alan basit (ama ne yazık ki uygulaması zor!) bir yaklaşımın en uygun politika olduğunu ve mümkün olan en az önbellek hatasıyla sonuçlandığını gösterdi.

TÜYO: EN UYGUN ile KARŞILAŞTIRMAK YARARLIDIR

(TIP: COMPARING AGAINST OPTIMAL IS USEFUL)

En uygun politikası, gerçek bir politika olarak çok pratik olmasa da, simülasyon veya diğer çalışmalarda bir karşılaştırma noktası olarak inanılmaz derecede faydalıdır. Şaşaalı yeni algoritmanızın %80 isabet oranına sahip olduğunu söylemek, tek başına anlamlı değildir; ancak en uygunda %82 isabet oranına ulaştığını söylemek (ve böylece yeni yaklaşımınız en uyguna oldukça yakın olur) sonucu daha anlamlı hale getirir ve ona bağlam verir. Böylece, yaptığınız herhangi bir çalışmada, en uygunun ne olduğunu bilmek daha iyi bir karşılaştırma yapmanızı sağlar, ne kadar iyileştirmenin hala mümkün olduğunu ve ayrıca ideale yeterince yakın olduğu için politikanızı daha iyi hale getirmeyi ne zaman bırakabileceğinizi gösterir. [AD03].

Umarım, en uygun politikasının arkasındaki sezgi mantıklıdır. Şöyle düşünün: Bazı sayfaları atmanız gerekiyorsa, neden şu an ihtiyacınız olandan en uzağını atmıyorsunuz? Bunu yaparak, aslında önbellekteki diğer tüm sayfaların uzaktaki sayfalardan daha önemli olduğunu söylüyorsunuz. Bunun doğru olmasının nedeni basit: En uzaktaki sayfaya geçmeden önce diğer sayfalara bakacaksınız.

En uygun politikasının verdiği kararları anlamak için basit bir örnek üzerinden gidelim. Bir programın aşağıdaki sanal sayfa akışına eriştiğini varsayalım: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1. Şekil 22.1 önbelleğin 3 sayfaya sığdığını varsayarak en uygunun davranışını göstermektedir.

Şekilde, aşağıdaki eylemleri görebilirsiniz. Önbellek boş bir durumda başladığından, ilk üç erişimin kaçırılması şaşırtıcı değildir; böyle bir ıskalama bazen **soğuk başlangıç ıskalama** veya **zorunlu ıskalama (cold start miss or compulsory miss)** olarak anılır. Ardından yeniden önbellekte isabet olmuş olan 0 ve 1.sayfalara bakarız. Sonunda başka bir ıskalamaya (sayfa 3'e) ulaşıyoruz, ancak bu sefer önbellek dolu; bir değiştirme yapılmalı! Ki şu soruyu akla getiriyor: hangi sayfayı değiştirmeliyiz? En uygun politikası ile, o anda önbellekte bulunan her sayfanın (0, 1 ve 2) geleceğini inceleriz ve gelecekte 0'a neredeyse anında, 1'e az daha sonra ve 2'ye de en uzak konumda erişildiğini görürüz. Bu yüzden, optimal politikanın kolay bir seçimi vardır: sayfa 2'nin çıkarılması, önbellekte 0, 1 ve 3. sayfalarla sonuçlanır.

Erişim	İsabet/Kayıp?	Çıkartma	Ortaya Çıkan Önbellek Durumu
0	Kayıp		0
1	Kayıp		0, 1
2	Kayıp		0, 1, 2
0	İsabet		0, 1, 2
1	İsabet		0, 1, 2
3	Kayıp	2	0, 1, 3
0	İsabet		0, 1, 3
3	İsabet		0, 1, 3
1	İsabet		0, 1, 3
2	Kayıp	3	0, 1, 2
1	İsabet		0, 1, 2

Şekil 22.1: **En Uygun Politikasının İzlenmesi (Tracing The Opimal Policy)**

ÖTE YANDAN: ÖNBELLEK KAYBI TÜRLERİ

(ASIDE: TYPES OF CACHE MISSES)

Bilgisayar mimarisi dünyasında, mimarlar bazen eksiklikleri türlerine göre üç kategoriye ayırmayı yararlı bulurlar: zorunlu, kapasiteli ve bazen Üç C [H87] olarak da adlandırılan çakışma kayıpları. Önbellek başlangıçta boş olduğundan ve bu, öğeye ilk referans olduğundan, **zorunlu bir hata (veya soğuk başlatma hatası [EF78])** (**A compulsory miss (or cold-start miss))** meydana gelir; Buna karşılık, önbellekte yer kalmadığı ve önbelleğe yeni bir öğe getirmek için bir öğeyi çıkarmak zorunda kaldığı için bir **kapasite kaybı (capacity miss)** da meydana gelir. Kaybın üçüncü türü (bir **çakışma hatası (conflict miss)**), **kümesel ilişkilendirme (set- associativity)** olarak bilinen bir şey nedeniyle bir öğenin donanım önbelleğinde nereye yerleştirilebileceğine ilişkin sınırlamalarla donanımda ortaya çıkar; bu durum İşletim Sistemi sayfa önbelleğinde ortaya çıkmaz, çünkü bu tür önbellekler her zaman **tam olarak ilişkilendirilebilir (fully associative),** yani bir sayfanın bellekte nereye yerleştirilebileceği konusunda herhangi bir kısıtlama yoktur. Detaylar için H&P'ye göz atınız [HP06].

Sonraki üç referans isabettir, ancak daha sonra uzun zaman önce çıkardığımız 2. sayfaya geliyoruz ve bir kez daha ıskalıyoruz. Burada en uygun politikası, önbellekteki (0, 1 ve 3) her sayfanın geleceğini tekrar inceler ve 1. sayfayı (erişilmek üzere olan) çıkarmadığı sürece sorun olmayacağını görür. Örnek, 3. sayfanın çıkarıldığını gösteriyor, ancak 0 da iyi bir seçim olabilirdi. Son olarak 1. sayfaya geliyoruz ve iz sürme tamamlanıyor.

Önbellek için isabet oranını da hesaplayabiliriz: 6 isabet ve 5 kayıp için isabet oranı Hits/Hits+Misses yani 6/6+5 ya da 54.5%'dir. Ayrıca, %85,7'lik bir isabet oranıyla sonuçlanan, modulo zorunlu kaçırmaları (yani, belirli bir sayfadaki ilk ıskalamayı dikkate almama) isabet oranını da hesaplayabilirsiniz.

Ne yazık ki, planlama politikalarının geliştirilmesinde gördüğümüz gibi, gelecek genel olarak bilinmemektedir; genel amaçlı bir işletim sistemi 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 politikası yalnızca bir karşılaştırma noktası olarak, "mükemmel" ifadesine ne kadar yakın olduğumuzu bilmek için hizmet verir.

22.3 Basit Bir Politika: İlk Giren İlk Çıkar (A Simple Policy: FIFO)

Birçok eski sistem, en yakına yaklaşmaya çalışmanın karmaşıklığından kaçındı ve çok basit değiştirme politikaları kullandı. Örneğin, bazı sistemler sayfaların sisteme girdiklerinde basitçe bir kuyruğa yerleştirildiği FIFO (ilk giren ilk çıkar) değişimini kullandı; bir değiştirme gerçekleştiğinde, sıranın sonundaki sayfa ("ilk giren" sayfası) çıkarılır. FIFO'nun büyük bir gücü var:

Erişim	İsabet/Kayıp?	Çıkartma	Ortaya Çıkan Önbellek Durumu
0	Kayıp		İlk giren→ 0
1	Kayıp		İlk giren → 0, 1
2	Kayıp		İlk giren → 0, 1, 2
0	İsabet		İlk giren → 0, 1, 2
1	İsabet		İlk giren → 0, 1, 2
3	Kayıp	0	İlk giren → 1, 2, 3
0	Kayıp	1	İlk giren → 2, 3, 0
3	İsabet		İlk giren → 2, 3, 0
1	Kayıp	2	İlk giren → 3, 0, 1
2	Kayıp	3	İlk giren → 0, 1, 2
1	İsabet		İlk giren → 0, 1, 2

Şekil 22.2: **FIFO Politikasının İzlenmesi** (**Tracing The FIFO Policy)**

uygulanması oldukça basit. Örnek referans akışımızda FIFO'nun nasıl çalıştığını inceleyelim (Şekil 22.2, sayfa 5). İzlemeye tekrar üç zorunlu ıskalama ile başlıyoruz. 0,1 ve 2. sayfalar ve ardından hem 0 hem 1. sayfalar isabet ediyor.

[1]If you can, let us know! We can become rich together. Or, like the scientists who "discovered" cold fusion, widely scorned and mocked [FP89].

Ardından, sayfa 3'e başvurulur ve bu da bir hataya neden olur; FIFO ile değiştirme kararı kolaydır: "ilk" olan sayfayı seçin (şekildeki önbellek durumu, ilk giren sayfa solda olacak şekilde FIFO düzeninde tutulur), yani sayfa 0'ı seçilirNe yazık ki, bir sonraki erişimimiz 0. sayfaya olacak, bu da başka bir ıskalamaya ve (1. sayfanın) değiştirilmesine neden oluyor. Daha sonra 3. sayfaya isabet ederiz, ancak 1 ve 2'yi kaçırırız ve sonunda 1'e isabet ederiz.

FIFO'yu optimumla karşılaştırırsak, FIFO önemli ölçüde daha kötüdür: %36,4'lük bir isabet oranı (veya zorunlu ıskalamalar hariç %57,1). FIFO, blokların önemini basitçe belirleyemez: 0. sayfaya birkaç kez erişilmiş olsa da, FIFO, hafızaya ilk getirilen sayfa olduğu için yine de onu atar.

ÖTE YANDAN: BELADY'NİN ANOMALİSİ

(ASIDE: BELADY'S ANOMALY)

Belady (en uygun politikasından) ve meslektaşları, biraz beklenmedik şekilde davranan ilginç bir referans akışı buldular [BNS69]. Bellek referans akışı: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. İnceledikleri değiştirme politikası FIFO idi. İlginç kısım: 3 sayfalık bir önbellek boyutundan 4 sayfaya geçerken önbellek isabet oranının nasıl değiştiğiydi.

Genel olarak, önbellek büyüdükçe önbellek isabet oranının artmasını (iyileşmesini) beklersiniz. Ancak bu durumda, FIFO ile daha da kötüye gidiyor! İsabet ve kayıpları hesaplayın ve kendiniz görün. Bu garip davranışa genellikle Belady'nin Anomalisi **(Belady's Anomaly)** (beraber yazdığı yazarların üzüntüsüne) denir.

LRU gibi diğer bazı politikalar bu sorundan muzdarip değildir. Nedenini tahmin edebilir misiniz? Görünüşe göre LRU, **yığın özelliği (stack property)** [M+70] olarak bilinen şeye sahip. Bu özelliğe sahip algoritmalar için, N + 1 boyutunda bir önbellek doğal olarak N boyutunda bir önbelleğin içeriğini içerir. Yani, önbellek boyutu artırılırken isabet oranı ya aynı kalacak ya da artacaktır. FIFO ve Rastgele (diğerleri arasında) açıkça stack özelliğine uymazlar ve bu nedenle anormal davranışlara karşı hassastırlar.

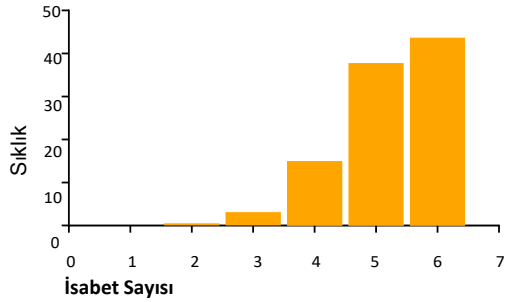
Erişim	İsabet/Kayıp?	Çıkartma	Önbellek Çıkan Önbellek Durumu
0	Kayıp		0
1	Kayıp		0, 1
2	Kayıp		0, 1, 2
0	İsabet		0, 1, 2
1	İsabet		0, 1, 2
3	Kayıp	0	1, 2, 3
0	Kayıp	1	2, 3, 0
3	İsabet		2, 3, 0
1	Kayıp	3	2, 0, 1
2	İsabet		2, 0, 1
1	İsabet		2, 0, 1

Şekil 22.3: **Random Politikasının İzlenmesi** (**Tracing The Random Policy)**

22.4 Bir Diğer Basit Politika: Rastgele (Another Simple Policy: Random)

Başka bir benzer değiştirme politikası, bellek baskısı altında değiştirmek için rastgele bir sayfa seçen Rastgele'dir. Rastgele, FIFO'ya benzer özelliklere sahiptir; uygulaması basittir, ancak hangi blokların tahliye edileceğini seçerken gerçekten çok akıllı olmaya çalışmaz. Ünlü örnek referans akışımızda Random'ın ne yaptığına bakalım. (Şekil 22.3'e bakın).

Elbette, Random'un ne yaptığı tamamen Random'un seçimlerinde ne kadar şanslı (veya şanssız) olduğuna bağlıdır. Yukarıdaki örnekte Random, FIFO'dan biraz daha iyi ve optimalden biraz daha kötüdür. Aslında Random deneyini binlerce kez çalıştırabilir ve genel olarak nasıl çalıştığını belirleyebiliriz Şekil 22.4, Random'ın her biri farklı bir rasgele parametreyle 10.000'den fazla denemede kaç isabet elde ettiğini gösterir. Gördüğünüz gibi, bazen (zamanın %40'ından biraz fazlasında) Rastgele, optimal kadar iyidir ve örnek iz üzerinde 6 isabet elde eder; bazen 2 veya daha az isabet elde ederek çok daha kötüsünü yapar. Rastgele'nin nasıl olduğu şansa bağlıdır.



Şekil 22.4: **Random Performance Over 10,000 Trials**

Access	Hit/Miss?	Evict	Resulting Cache State
0	Miss		LRU→ 0
1	Miss		LRU→ 0, 1
2	Miss		LRU→ 0, 1, 2
0	Hit		LRU→ 1, 2, 0
1	Hit		LRU→ 2, 0, 1
3	Miss	2	LRU→ 0, 1, 3
0	Hit		LRU→ 1, 3, 0
3	Hit		LRU→ 1, 0, 3
1	Hit		LRU→ 0, 3, 1
2	Miss	0	LRU→ 3, 1, 2
1	Hit		LRU→ 3, 2, 1

Figure 22.5: **Tracing The LRU Policy**

22.5 Geçmişi Kullanmak: LRU (Using History: LRU)

Ne yazık ki, FIFO veya Rastgele gibi basit bir politikanın ortak bir sorunu olması muhtemeldir: önemli bir sayfayı, tekrar başvurulmak üzere olan bir sayfayı atabilir. FIFO, ilk getirilen sayfayı atar; Bu, üzerinde önemli kod veya veri yapılarının bulunduğu bir sayfaysa, kısa süre sonra tekrar sayfaya alınacak olsa bile nasılsa atılır. Bu nedenle, FIFO, Rastgele ve benzeri politikaların en uyguna yaklaşması pek olası değildir; daha akıllı bir şeye ihtiyaç vardır.

Zamanlama politikasında yaptığımız gibi, geleceğe dair tahminimizi geliştirmek için bir kez daha geçmişe yaslanıyor ve tarihi rehberimiz olarak kullanıyoruz. Örneğin, bir program yakın geçmişte bir sayfaya eriştiyse, yakın gelecekte bu sayfaya tekrar erişmesi muhtemeldir.

Bir sayfa değiştirme politikasının kullanabileceği bir tür tarihsel bilgi **sıklıktır (frequency)**; bir sayfaya birçok kez erişildiyse, belki de bir değeri olduğu için değiştirilmemelidir. Bir sayfanın daha sık kullanılan bir özelliği, **yakın zamandaki (recency)** erişimidir; bir sayfaya ne kadar yakın zamanda erişildiyse, belki de o sayfaya tekrar erişilme olasılığı o kadar yüksektir.

Bu politikalar ailesi, insanların **yerellik ilkesi (principle of locality)** [D70] olarak adlandırdığı, temel olarak programlar ve davranışları hakkında sadece bir gözlem olan şeye dayanmaktadır. Bu ilkenin oldukça basit bir şekilde söylediği şey, programların belirli kod dizilerine (örneğin, bir döngüde) ve veri yapılarına (örneğin, döngü tarafından erişilen bir diziye) oldukça sık erişme eğiliminde olduğudur; bu nedenle, hangi sayfaların önemli olduğunu anlamak için geçmişi kullanmaya çalışmalı ve çıkartma zamanı geldiğinde bu sayfaları hafızamızda tutmalıyız.

Ve böylece, tarihsel tabanlı basit bir algoritma ailesi doğar. **En Az Sık Kullanılan (Least-Frequently-Used (LFU))** politikası, bir tahliyenin gerçekleşmesi gerektiğinde en az kullanılan sayfanın yerini alır. Benzer şekilde, **En Az Son Kullanılan (Least-Recently-Used (LRU))** ilkesi en son kullanılan sayfanın yerini alır. Bu algoritmaların hatırlanması kolaydır: Bir ismi bir kez öğrendiğinizde tam olarak ne işe yaradığını da bilirsiniz, bu da bir isim için mükemmel bir özelliktir

LRU'yu daha iyi anlamak için örnek referans sistemimizde LRU'nun nasıl çalıştığını inceleyelim.

ÖTE YANDAN: YERELLİK ÇEŞİTLERİ

(ASIDE: TYPES OF LOCALITY)

Programların sergileme eğiliminde olduğu iki tür yerellik vardır. İlki, bir P sayfasına erişilirse, etrafındaki sayfalara da (P - 1 veya P + 1 diyelim) büyük olasılıkla erişileceğini belirten **uzamsal konum (spatial locality)** olarak bilinir. İkincisi, yakın geçmişte erişilen sayfalara yakın gelecekte tekrar erişilebileceğini belirten **zamansal konumdur (temporal locality)**. Bu tür konumların var olduğu varsayımı, bu tür konumlar mevcut olduğunda programların hızlı çalışmasına yardımcı olmak için birçok düzeyde talimat, veri ve adres çevirisi önbelleğe alma uygulayan donanım sistemlerinin önbelleğe alma hiyerarşilerinde büyük bir rol oynar.

Elbette, sık sık adlandırıldığı şekliyle **yerellik ilkesi (principle of locality)**, tüm programların uyması gereken katı ve katı bir kural değildir. Aslında, bazı programlar belleğe (veya diske) oldukça rastgele bir şekilde erişir ve erişim akışlarında çok fazla veya herhangi bir yer göstermez. Bu nedenle, herhangi bir türden (donanım veya yazılım) önbellek tasarlarken yerellik akılda tutulması gereken iyi bir şey olsa da, başarıyı garanti etmez. Daha ziyade, bilgisayar sistemlerinin tasarımında sıklıkla yararlı olduğu kanıtlanan bir buluşsaldır.

Şekil 22.5 (sayfa 7) sonuçları göstermektedir. Şekilden, LRU'nun Random veya FIFO gibi durum bilgisi olmayan politikalardan daha iyisini yapmak için geçmişi nasıl kullanabileceğini görebilirsiniz. Örnekte LRU, bir sayfayı ilk kez değiştirmesi gerektiğinde 2. sayfayı çıkarır, çünkü 0 ve 1'e daha yakın zamanda erişilmiştir. Ardından, 1 ve 3'e daha yakın zamanda erişildiği için sayfa 0'I değiştirir. Her iki durumda da, LRU'nun tarihe dayalı kararı doğru çıkıyor ve sonraki referanslar isabetli oluyor. Böylece örneğimizde, LRU mümkün olduğu kadar en iyisini yapar, performansında en uygun olanı eşleştirir. Ayrıca, bu algoritmaların karşıtlarının da bulunduğunu not etmeliyiz: **En Sık Kullanılan (Most-Frequently-Used (MFU))** ve **En Son Kullanılan (Most-Recently-Used (MRU))**. Çoğu durumda (hepsinde değil!), bu politikalar iyi çalışmaz çünkü çoğu programın sergilediği yerelliği benimsemek yerine onu görmezden gelirler.

22.6 İş Yükü Örnekleri (Workload Examples)

Bu politikaların bazılarının nasıl davrandığını daha iyi anlamak için birkaç örneğe daha bakalım. Burada küçük izler yerine daha karmaşık **iş yüklerini (work-loads)** inceleyeceğiz. Ancak bu iş yükleri bile büyük ölçüde basitleştirilmiştir; daha iyi bir çalışma, uygulama izlerini içerecektir.

İlk iş yükümüzün yerelliği yoktur, bu da her referansın erişilen sayfalar kümesindeki rastgele bir sayfaya referans olduğu anlamına gelir. Bu basit örnekte, iş yükü, başvurulacak bir sonraki sayfayı rastgele seçerek, zaman içinde 100 benzersiz sayfaya erişir; toplamda 10.000 sayfaya erişildi. Deneyde, her bir politikanın önbellek boyutları aralığında nasıl davrandığını görmek için önbellek boyutunu çok küçükten (1 sayfa) tüm benzersiz sayfaları tutmaya yetecek kadar (100 sayfa) değiştirdik.

[2]OK, we cooked the results. But sometimes cooking is necessary to prove a point.

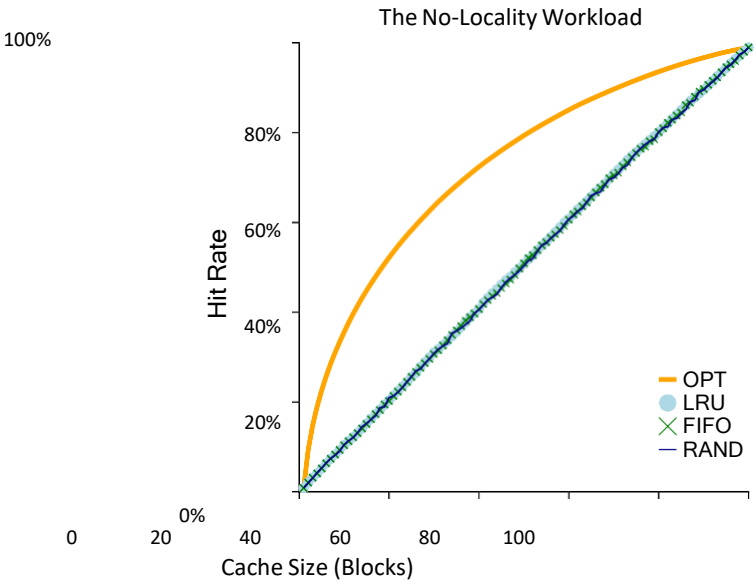


Figure 22.6: **The No-Locality Workload**

Şekil 22.6 en uygun, LRU, Rastgele ve FIFO için deneyin sonuçlarını göstermektedir. Şeklin y ekseni, her bir politikanın ulaştığı isabet oranını gösterir; x ekseni, yukarıda açıklandığı gibi önbellek boyutunu belirtir.

Grafikten bir takım sonuçlar çıkarabiliriz. İlk olarak, iş yükünde yerellik olmadığında, hangi gerçekçi politikayı kullandığınızın pek bir önemi yoktur; LRU, FIFO ve Random, tam olarak önbelleğin boyutuna göre belirlenen isabet oranıyla aynı performansı gösterir. İkinci olarak, önbellek tüm iş yükünü sığdıracak kadar büyük olduğunda, hangi politikayı kullandığınız da önemli değildir; başvurulan tüm bloklar önbelleğe sığdığında tüm politikalar (Rastgele bile) %100 isabet oranına yakınsar. Son olarak, optimalin gerçekçi politikalardan belirgin şekilde daha iyi performans gösterdiğini görebilirsiniz; Geleceğe göz atmak, eğer mümkünse, çok daha iyi bir değiştirme işi yapar.

İnceleyeceğimiz bir sonraki iş yükü, yerellik sergileyen “80-20” iş yükü olarak adlandırılıyor: referansların %80'i, sayfaların %20'sine (“sıcak” sayfalar) yapılıyor; referansların kalan %20'si, sayfaların geri kalan %80'ine (“soğuk” sayfalar) yapılmıştır. İş yükümüzde yine toplam 100 tekil sayfa var; bu nedenle, çoğu zaman "sıcak" sayfalara, geri kalan zamanlarda "soğuk" sayfalara atıfta bulunulur. Şekil 22.7 (sayfa 10), ilkelerin bu iş yüküyle nasıl performans gösterdiğini gösterir.

Şekilden de görebileceğiniz gibi, hem rasgele hem de FIFO oldukça iyi iş çıkarsa da, sıcak sayfaları tutma olasılığı daha yüksek. LRU daha iyi iş çıkarıyor; bu sayfalara geçmişte sıklıkla atıfta bulunulduğu için yakın gelecekte tekrar atıfta bulunulması muhtemeldir. En uygun bir kez daha iyi bir iş çıkarıyor, bu da bize LRU'nun tarihsel bilgisinin mükemmel olmadığını gösterir.

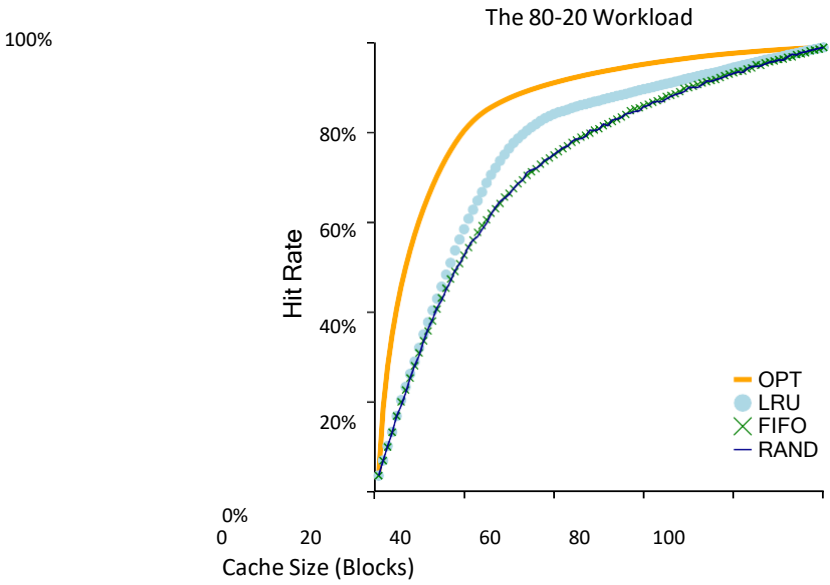


Figure 22.7: **The 80-20 Workload**

Şimdi merak ediyor olabilirsiniz: LRU'nun Random ve FIFO üzerindeki gelişimi gerçekten o kadar önemli mi? Cevap, her zamanki gibi, "duruma göre değişir". Her ıskalamanın maliyeti çok yüksekse (nadir değil), o zaman isabet oranındaki küçük bir artış (ıskalama oranında azalma) bile performansta büyük bir fark yaratabilir. Hatalar o kadar maliyetli değilse, o zaman elbette LRU ile mümkün olan faydalar neredeyse o kadar önemli değildir.

Son bir iş yüküne bakalım. Buna "döngüsel sıralı" iş yükü diyoruz, içinde olduğu gibi, 0, sonra 1, ...'den başlayıp 49. sayfaya kadar sırayla 50 sayfaya atıfta bulunuyoruz ve sonra bu erişimleri tekrarlayarak döngü oluşturuyoruz. 50 benzersiz sayfaya toplam 10.000 erişim için. Şekil 22.8'deki son grafik, politikaların bu iş yükü altındaki davranışını göstermektedir.

Pek çok uygulamada (veritabanları [CD85] gibi önemli ticari uygulamalar dahil) yaygın olan bu iş yükü, hem LRU hem de FIFO için en kötü durumu temsil eder. Bu algoritmalar, döngüsel sıralı bir iş yükü altında eski sayfaları atar; ne yazık ki iş yükünün döngüsel yapısı nedeniyle bu eski sayfalara, politikaların önbellekte tutmayı tercih ettiği sayfalardan daha erken erişilecektir. Gerçekten de, 49 boyutunda bir önbellekle bile, 50 sayfalık döngüsel sıralı bir iş yükü, %0 isabet oranıyla sonuçlanır. İlginç bir şekilde, Rastgele fark edilir ölçüde daha iyidir, en uyguna pek yaklaşmıyor ama en azından sıfır olmayan bir isabet oranına

ulaşıyor. Random'ın bazı güzel özellikleri olduğu ortaya çıktı; böyle bir özellik, tuhaf köşe durum davranışlarına sahip olmamaktır.

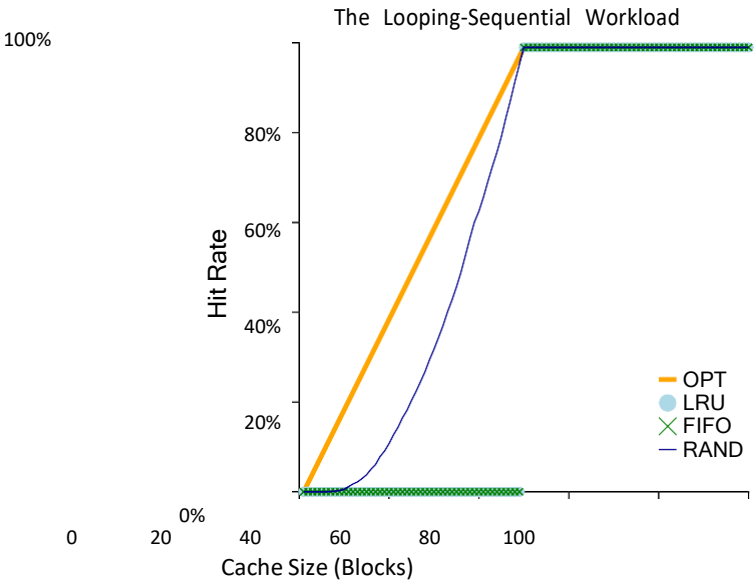


Figure 22.8: **The Looping Workload**

22.7 Tarihsel Algoritmaların Uygulanması (Implementing Historical Algorithms)

Gördüğünüz gibi, LRU gibi bir algoritma, genellikle FIFO veya Random gibi önemli sayfaları atabilen daha basit politikalardan daha iyi bir iş çıkarabilir. Ne yazık ki, tarihsel politikalar bize yeni bir meydan okuma sunuyor: onları nasıl uygularız?

Örneğin, LRU'yu ele alalım. Mükemmel bir şekilde uygulamak için çok çalışmamız gerekiyor. Spesifik olarak, her sayfa erişiminde (yani, her bellek erişimi, ister bir komut getirme, ister yükleme veya depolama), bu sayfayı listenin önüne (yani, MRU tarafına) taşımak için bazı veri yapılarını güncellememiz gerekir. Bunu, FIFO sayfa listesine yalnızca bir sayfa çıkarıldığında (ilk giren sayfasını kaldırarak) veya listeye yeni bir sayfa eklendiğinde (son giren tarafına) erişildiği FIFO ile karşılaştırın. En az ve en son hangi sayfaların kullanıldığını takip etmek için, sistemin her bellek referansında bazı hesaplama çalışmaları yapması gerekir. Açıkçası, büyük bir özen gösterilmediği takdirde, bu tür bir muhasebe performansı büyük ölçüde azaltabilir.

Bunu hızlandırmaya yardımcı olabilecek bir yöntem, biraz donanım desteği eklemektir. Örneğin, bir makine, her sayfa erişiminde, bellekteki bir zaman alanını güncelleyebilir (örneğin bu işlem başına sayfa tablosunda veya sistemdeki fiziksel sayfa başına bir girişle

bellekteki ayrı bir dizide olabilir). Böylece bir sayfaya girildiğinde zaman alanı donanım tarafından güncel zamana ayarlanacaktır. Ardından, bir sayfayı değiştirirken işletim sistemi, en son kullanılan sayfayı bulmak için sistemdeki tüm zaman alanlarını tarayabilir.

Ne yazık ki, bir sistemdeki sayfa sayısı arttıkça, yalnızca en son kullanılan mutlak sayfayı bulmak için çok sayıda kez tarama yapmak çok pahalıdır. 4 KB sayfalara bölünmüş, 4 GB belleğe sahip modern bir makine hayal edin. Bu makinenin 1 milyon sayfası vardır ve bu nedenle LRU sayfasını bulmak, modern CPU hızlarında bile uzun zaman alacaktır. Bu daşu soruyu akla getiriyor: değiştirilecek mutlak en eski sayfayı gerçekten bulmamız gerekiyor mu? Bunun yerine bir yaklaşımla kurtulabilir miyiz?

CAN ALICI NOKTA: LRU YER DEĞİŞTİRME POLİTİKASINI NASIL KULLANIRIZ

(CRUX: HOW TO IMPLEMENT AN LRU REPLACEMENT POLICY)

Mükemmel LRU'yu uygulamanın pahalı olacağı göz önüne alındığında, buna bir şekilde yaklaşabilir ve yine de istenen davranışı elde edebilir miyiz?

22.8 Yaklaşık LRU (Approximating LRU)

Görünüşe göre cevap evet: LRU'ya yaklaşmak, hesaplama yükü açısından daha makul ve aslında birçok modern sistemin yaptığı da bu. Fikir, bir **kullanım biti (use bit)** (bazen **referans biti (reference bit)** olarak da adlandırılır) biçiminde bir miktar donanım desteği gerektirir; bunlardan ilki, çağrılı ilk sistem olan Atlas tek düzey deposunda [KE+62] uygulandı. Sistemin her sayfası için bir kullanım biti vardır ve kullanım bitleri bellekte bir yerde yaşar (örneğin, işlem başına sayfa tablolarında veya herhangi bir yerde bir dizide olabilirler). Bir sayfaya başvuru yapıldığında (yani okunduğunda veya yazıldığında), kullanım biti donanım tarafından 1'e ayarlanır. Ancak donanım asla biti temizlemez (yani 0'a ayarlar); bu OS'nin sorumluluğudur.

İşletim Sistemi, LRU'ya yaklaşmak için kullanım bitini nasıl kullanır? Pek çok yol olabilir, ancak **saat algoritması (clock algorithm)** [C69] ile basit bir yaklaşım önerildi. Dairesel bir liste halinde düzenlenmiş sistemin tüm sayfalarını hayal edin. Bir **saat ibresi (clock hand)**, başlamak için belirli bir sayfayı işaret eder (hangisi olduğu gerçekten önemli değildir). Bir değiştirmenin olması gerektiğinde, İşletim Sistemi şu anda işaret edilen sayfa P'nin kullanım bitinin 1 veya 0 olup olmadığını kontrol eder. Böylece, P için kullanım biti 0'a ayarlanır (temizlenir) ve saat ibresi bir sonraki sayfaya (P + 1) yükseltilir. Algoritma, 0'a ayarlanmış bir kullanım biti bulana kadar devam eder; bu, bu sayfanın yakın zamanda kullanılmadığını (veya en kötü durumda, tüm sayfaların kullanılmış olduğunu ve şimdi tüm sayfa setinde arama yaptığımızı ima eder, ardından tüm bitleri temizler).

Bu yaklaşımın, LRU'ya yaklaşmak için bir kullanım biti kullanmanın tek yolu olmadığına dikkat edin. Aslında, kullanım bitlerini periyodik olarak temizleyen ve ardından hangi sayfaların 1'e karşı 0 kullanım bitlerine sahip olduğunu ayırt eden herhangi bir yaklaşım, hangisinin değiştirileceğine karar vermek için iyi olacaktır. Corbato'nun saat algoritması, biraz başarılı olan ve kullanılmayan bir sayfayı aramak için tüm belleği tekrar tekrar taramama gibi güzel bir özelliğe sahip olan erken yaklaşımlardan yalnızca biriydi.

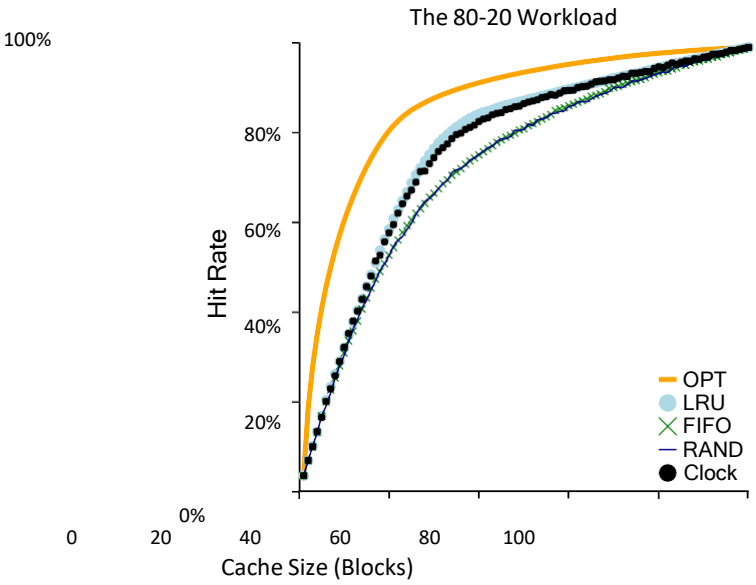


Figure 22.9: **The 80-20 Workload With Clock**

Bir saat algoritma varyantının davranışı Şekil 22.9'da gösterilmiştir. Bu değişken, değiştirme işlemi yapılırken sayfaları rasgele tarar; referans biti 1'e ayarlanmış bir sayfayla karşılaştığında, biti temizler (yani, 0'a ayarlar); referans biti 0'a ayarlanmış bir sayfa bulduğunda, onu kurban olarak seçer. Gördüğünüz gibi, mükemmel bir LRU kadar iyi iş görmese de, tarihi hiç dikkate almayan yaklaşımlardan daha iyi iş çıkarıyor.

22.9 Kirli Sayfaları Dikkate Almak (Considering Dirty Pages

Saat algoritmasında yaygın olarak yapılan küçük bir değişiklik (orijinal olarak Corbato [C69] tarafından önerilmiştir), bir sayfanın belleketyken **değiştirilip (modified)** değiştirilmediğine ilişkin ek değerlendirmedir. Bunun nedeni: Bir sayfa değiştirilmişse ve bu nedenle **kirliyse (dirty)**, onu çıkarmak için diske geri yazılması gerekir ki bu maaliyetlidir. Değiştirilmemişse (ve dolayısıyla **temizse (clean)**), bu değiştirme ücretsizdir; fiziksel çerçeve, ek Giriş/Çıkış olmaksızın başka amaçlar için kolayca yeniden kullanılabilir. Bu nedenle, bazı sanal makine sistemleri kirli sayfalar yerine temiz sayfaları çıkarmayı tercih eder.

Bu davranışı desteklemek için, donanım **değiştirilmiş bir bit (modified bit)** (diğer adıyla **kirli bit (dirty bit)**) içermelidir. Bu bit, herhangi bir sayfa yazıldığında ayarlanır ve bu nedenle sayfa değiştirme algoritmasına dahil edilebilir. Örneğin saat algoritması, önce hem kullanılmayan hem de temiz sayfaları çıkartmak için taramak üzere değiştirilebilir; bunları bulamazsa, ardından kullanılmayan kirli sayfalar için vb.

22.10 Diğer Sanal Makine Politikaları (Other VM Policies)

Sayfa değiştirme (page selection), sanal makine alt sisteminin kullandığı tek politika değildir (ancak en önemlisi olabilir). Örneğin, işletim sisteminin bir sayfanın ne zaman belleğe alınacağına da karar vermesi gerekir. Bazen sayfa seçimi ilkesi olarak adlandırılan bu ilke (Denning [D70] tarafından adlandırıldığı gibi), işletim sistemine bazı farklı seçenekler sunar.

Çoğu sayfa için, işletim sistemi yalnızca **isteğe bağlı sayfalamayı (demand paging)** kullanır; bu, işletim sisteminin sayfayı erişildiğinde "istek üzerine" olduğu gibi belleğe getirmesi anlamına gelir. Tabii ki, işletim sistemi bir sayfanın kullanılmak üzere olduğunu tahmin edebilir ve bu nedenle onu önceden getirebilir; bu davranış **önceden getirme (prefetching)** olarak bilinir ve yalnızca makul bir başarı şansı olduğunda yapılmalıdır. Örneğin, bazı sistemler, bir P kod sayfası belleğe getirilirse, bu P +1 kod sayfasına büyük olasılıkla yakında erişileceğini ve bu nedenle de belleğe alınması gerektiğini varsayar.

Başka bir ilke, işletim sisteminin sayfaları diske nasıl yazacağını belirler. Tabii ki, her seferinde bir tane yazılabilirler; bununla birlikte, birçok sistem bunun yerine bellekte bir dizi bekleyen yazma işlemini toplar ve bunları bir (daha verimli) yazma işleminde diske yazar. Bu davranışa genellikle yazma işlemlerinin **kümelenmesi (clustering)** veya basitçe **gruplandırılması (grouping)** adı verilir ve tek bir büyük yazmayı birçok küçük olandan daha verimli şekilde gerçekleştiren disk sürücülerinin doğası nedeniyle etkilidir.

22.11 Çarpma (Thrashing)

Bitirmeden önce, son bir soruyu ele alıyoruz: Bellek aşırı yüklendiğinde ve çalışan işlemler kümesinin bellek talepleri mevcut fiziksel belleği aştığında işletim sistemi ne yapmalıdır? Bu durumda, sistem sürekli çağrı yapacaktır, bu durum bazen **çarpma (thrashing)** [D70] olarak adlandırılır.

Daha önceki bazı işletim sistemleri, meydana geldiğinde çarpma olayını hem tespit etmek hem de bununla başa çıkmak için oldukça karmaşık mekanizmalara sahipti. Örneğin, bir dizi işlem verildiğinde, bir sistem, azaltılmış işlem kümesinin **çalışma kümelerinin (working sets)** (aktif olarak kullandıkları sayfalar) belleğe sığması umuduyla, bir işlem alt kümesini çalıştırmamaya karar verebilir. Genel olarak **kabul kontrolü (admission control)** olarak bilinen bu yaklaşım, bazen her şeyi kötü bir şekilde yapmaya çalışmaktansa daha az işi iyi yapmanın daha iyi olduğunu belirtir, bu durum gerçek hayatta ve modern bilgisayar sistemlerinde (ne yazık ki) sıklıkla karşılaştığımız bir durumdur.

Bazı mevcut sistemler, aşırı bellek yüklemesine daha acımasız bir yaklaşım benimsiyor. Örneğin, Linux'un bazı sürümleri, belleğe gereğinden fazla talep olduğunda **yetersiz bellek öldürücü (out-of-memory-killer)** çalıştırır; bu arka plan programı, bellekte yoğun bir süreç seçer ve onu öldürür, böylece çok ince olmayan bir şekilde belleği azaltır. Bellek baskısını azaltmada başarılı olsa da, bu yaklaşım, örneğin X sunucusunu öldürürse ve dolayısıyla ekranı gerektiren

uygulamaları kullanılamaz hale getirirse, sorunlara yol açabilir.

22.1 Özet (Summary)

Tüm modern işletim sistemlerinin VM alt sisteminin bir parçası olan bir dizi sayfa değiştirme (ve diğer) politikalarının uygulamaya konduğunu gördük. Modern sistemler, saat gibi basit LRU yaklaşımlarına bazı ince ayarlar ekler; örneğin **tarama direnci (scan resistance)**, ARC [MM03] gibi birçok modern algoritmanın önemli bir parçasıdır. Taramaya dirençli algoritmalar genellikle LRU benzeridir ancak aynı zamanda döngüsel sıralı iş yükünde gördüğümüz LRU'nun en kötü davranışından kaçınmaya çalışır. Böylece, sayfa değiştirme algoritmalarının gelişimi devam ediyor.

Bellek erişimi ve disk erişimi süreleri arasındaki tutarsızlık çok büyük olduğundan, uzun yıllar boyunca değiştirme algoritmalarının önemi azalmıştı. Spesifik olarak, diske sayfalama çok pahalı olduğundan, sık sık sayfalamanın maliyeti engelleyiciydi; Basitçe söylemek gerekirse, değiştirme algoritmanız ne kadar iyi olursa olsun, sık sık değiştirme yapıyorsanız, sisteminiz dayanılmaz bir şekilde yavaşlar. Bu nedenle, en iyi çözüm basitti (entelektüel olarak tatmin edici değilse de): daha fazla bellek satın alın.

Ancak, çok daha hızlı depolama aygıtlarındaki (örneğin, Flash tabanlı SSD'ler ve Intel Optane ürünleri) son yenilikler bu performans oranlarını bir kez daha değiştirerek sayfa değiştirme algoritmalarında bir rönesansa yol açtı. Bu alandaki son çalışmalar için [SS10,W+21]'e bakın.

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. Belady. 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 tailorpolicies 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. Communications 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. Sumner. 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 propertymight 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 Storage 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.*

Ödev (Simulasyon) (Homework (Simulation))

Bu simulator, paging-policy.py, farklı sayfa değiştirme politikalarıyla uğraşabilmenizi sağlar. Detaylar için README'yi okuyun.

Sorular

1. Aşağıdaki bağımsız değişkenlerle rastgele adresler oluşturun: -s 0 -n 10, -s 1 -n 10 ve -s 2 -n 10. Politikayı FIFO'dan LRU'ya ve OPT (En Uygun Politikası)'ye değiştirin. Söz konusu adres izlerindeki her erişimin isabet mi yoksa kayıp mı olduğunu hesaplayın.
2. 5 boyutunda bir önbellek için, aşağıdaki politikaların her birine en kötü durum adres referans akışları oluşturun: FIFO, LRU ve MRU en kötü durum referans akışları, olası en fazla ıskalamaya neden olur. En kötü durum referans akışları için, performansı önemli ölçüde artırmak ve OPT'ye (En uyguna) yaklaşmak için bir önbelleğe ihtiyaç var mı?
3. Rastgele bir iz oluşturun (python veya perl kullanın). Böyle bir izde farklı politikaların nasıl performans göstermesini beklersiniz?

   Çözüm:

   Kod (Python):

```
import random

def generateTrace(length):
  trace = [ ]
  for i in range(length):
    trace.append(random.randint(0, 75))
  return trace
trace = generateTrace(10)
print(trace)
```

   Açıklama:

   Bir önbelleğin performansı, izdeki belirli erişim modeline ve önbelleğin boyutuna bağlı olduğundan, farklı politikaların belirli bir iz üzerinde tam olarak nasıl performans göstereceğini tahmin etmek zordur. Ancak genel olarak konuşacak olursak en uygun politikasının en iyi performansı

göstermesi beklenir. Ardından LRU ve onun da ardından FIFO şeklinde bu sıralama devam eder. En uygun politikası, gelecekte en uzun süre kullanılmayacak olan öğenin önbellekten çıkarılmak üzere seçilmesiyle çalışır, böylece önbelleğin en verimli şekilde kullanılmasını sağlar. LRU politikası, önbellekten çıkarılmak üzere en uzun süre kullanılmamış olan öğeyi seçerek çalışır, FIFO politikası, önbellekte en uzun süre çıkarılmış olan öğeyi seçerek çalışır.

4. Şimdi biraz yerellik kullanarak bir iz oluşturun. Böyle bir izi nasıl oluşturabilirsiniz? LRU bunun üzerinde nasıl bir performans sergiler? RAND, LRU'dan ne kadar daha iyi? CLOCK nasıl çalışır? Peki ya farklı sayıların clock bitlerine sahip CLOCK?

Çözüm:

Kod (Python):

```
import random

def generateTraceWithLocality(length, localitySize):
  trace = [ ]
  start = random.randint(0, length - localitySize - 1)
  for i in range(start, start + localitySize):
    trace.append(i)
  for i in range(length - localitySize):
    trace.append(random.randint(0, length - 1))
  return trace
trace = generateTraceWithLocality(10, 3)
print(trace)
```

Açıklama:

Yerelliğe sahip bir izlemede, LRU iyi performans gösterir çünkü yakın zamanda kullanılan öğeleri önbellekte tutar ve gelecekte bunlara daha hızlı erişim sağlar. Yerelliğe sahip bir izlemede, RAND, öğelerin son kullanımını dikkate almadığından ve yakında tekrar kullanılması muhtemel öğeleri kaldırabileceğinden, LRU kadar iyi performans göstermez. Önbelleğe yeni bir öğe eklenmesi gerektiğinde ve bu öğe dolduğunda, ilke, saat ibresinin işaret ettiği öğeyi, bu öğe yakın zamanda

kullanılmadıysa, kaldırır. Bu durumda, öğeye "ikinci bir şans verilir" ve saat ibresi bir sonraki öğeye geçer. Yakın zamanda kullanılan öğelere önbellekte kalmaları için ikinci bir şans verdiği ve gelecekte bu öğelere daha hızlı erişim sağladığı için yerellik içeren bir izlemede CLOCK iyi performans gösterir.

CLOCK, saat bitlerini kullanarak çalıştığı için farklı numaralar için oluşturuşan saat bitlerinin sayısı CLOCK'un performansını etkiler. Saat bitleriyse önbellekteki her öğe için son kullanım bilgilerini depolamak için kullanılır. Ne kadar çok saat biti kullanılırsa, son kullanım bilgisi o kadar kesin olur, ancak aynı zamanda onu depolamak daha maaliyetli olur. Genel olarak, daha fazla saat biti kullanmak, yerellik içeren bir izlemede CLOCK performansını artırabilir, ancak aynı zamanda ek yükü de artırır ve her zaman daha iyi performans sağlamayabilir.

5. Gerçek bir uygulama oluşturmak ve sanal bir sayfa referans akışı oluşturmak için valgrind gibi bir program kullanın. Örneğin, valgrind --tool=lackey --trace-mem=yes ls komutunu çalıştırmak, ls programı tarafından yapılan her talimatın ve veri referansının neredeyse eksiksiz bir referans izini verir. Bunu yukarıdaki simülatöre kullanışlı hale getirmek için, önce her bir sanal bellek referansını sanal bir sayfa numarası referansına dönüştürmeniz gerekir (uzaklığı maskeleyerek ve ortaya çıkan bitleri aşağı kaydırarak yapılır). İsteklerin büyük bir kısmını karşılamak için uygulama çalıştırmanız için ne kadar büyük bir önbelleğe ihtiyaç vardır? Önbelleğin boyutu artan çalışma kümesinin bir grafiğini çizin.