13

Soyutlama: Adres Alanları

İlk zamanlarda bilgisayar sistemlerini kurmak kolaydı. Neden diye mi soruyorsunuz? Çünkü kullanıcılar fazla bir şey beklemiyordu. Tüm bu baş ağrılarına yol açan, "kullanım kolaylığı", "yüksek performans", "güvenilirlik" vb. beklentileri olan lanet olası kullanıcılardır. Bir dahaki sefere bu bilgisayar kullanıcılarından biriyle karşılaştığınızda, neden oldukları tüm sorunlar için onlara teşekkür edin.

13.1 Erken Dönemdeki Sistemler

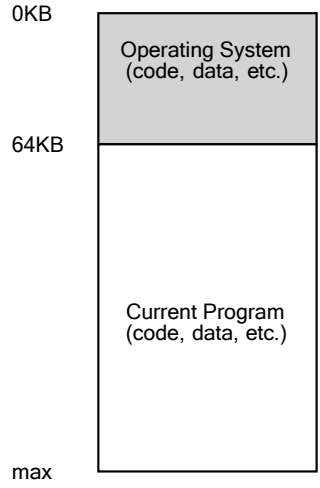
Bellek açısından bakıldığında, ilk makineler kullanıcılara fazla bir soyutlama sağlamıyordu. Temel olarak, makinenin fiziksel belleği (memory) Şekil 13.1'de gördüğünüz gibi bir şeye benziyordu.

İşletim sistemi bellekte yer alan (bu örnekte 0 fiziksel adresinden başlayan) bir dizi rutindi (aslında bir kütüphane) ve o anda fiziksel bellekte yer alan (bu örnekte 64k fiziksel adresinden başlayan) ve belleğin geri kalanını kullanan çalışan bir program (bir süreç) olurdu. Burada çok az yanılsama vardı ve kullanıcı işletim sisteminden fazla bir şey beklemiyordu. O günlerde işletim sistemi geliştiricileri için hayat kesinlikle kolaydı, değil mi?

13.2 Çoklu programlama ve Zaman Paylaşımı

Bir süre sonra makineler pahalı olduğu için, insanlar makineleri daha etkin bir şekilde paylaşmaya başladı. Böylece, birden fazla sürecin belirli bir zamanda çalışmaya hazır olduğu ve işletim sisteminin, örneğin biri G/Ç gerçekleştirmeye karar verdiğinde bunlar arasında geçiş yaptığı **çoklu programlama (multiprogramming)** dönemi doğdu [DV66]. Bunu yapmak CPU'nun etkin **kullanımını (utilization)** artırdı. Verimlilikteki bu tür artışlar, her bir makinenin yüz binlerce hatta milyonlarca dolara mal olduğu o günlerde özellikle önemliydi (ve siz Mac'inizin pahalı olduğunu düşünüyordunuz!).

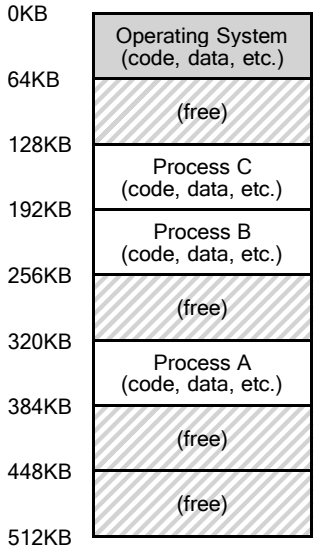
Ancak çok geçmeden insanlar makinelerden daha fazlasını talep etmeye başladı ve **zaman paylaşımı (time sharing)** çağı doğdu [S59, L60, M62, M83]. Özellikle, birçoğu toplu hesaplamanın özellikle programcıların kendileri üzerindeki sınırlamalarını fark etti [CV65],



Şekil 13.1: **İşletim Sistemleri: İlk Günlerimiz**

uzun (ve dolayısıyla etkisiz) program hata ayıklama döngülerinden bıkmışlardı. Birçok kullanıcı aynı anda bir makineyi kullanıyor olabileceğinden ve her biri o anda yürütülmekte olan görevlerinden zamanında yanıt beklediğinden (ya da umduğundan) etkileşim kavramı önemli hale geldi. Zaman paylaşımını uygulamanın bir yolu, bir süreci kısa bir süre çalıştırmak, tüm belleğe tam erişim vermek (Şekil 13.1), ardından durdurmak, tüm durumunu bir tür diske kaydetmek (tüm fiziksel bellek dahil), başka bir sürecin durumunu yüklemek, bir süre çalıştırmak ve böylece makinenin bir tür kaba paylaşımından bahsetmektedir [M+63]. Ne yazık ki, bu yaklaşımın büyük bir sorunu vardır: özellikle bellek büyüdükçe çok yavaştır. Kayıt seviyesi durumunu (PC, genel amaçlı kayıtlar, vb.) kaydetmek ve geri yüklemek nispeten hızlı olsa da belleğin tüm içeriğini diske kaydetmek acımasızca performanssızdır. Bu nedenle, süreçler arasında geçiş yaparken süreçleri bellekte bırakarak işletim sisteminin zaman paylaşımını verimli bir şekilde uygulamasına izin vermeyi tercih ederiz (Şekil 13.2'te gösterildiği gibi).

Diyagramda, üç süreç (A, B ve C) vardır ve her biri 512KB fiziksel belleğin küçük bir bölümünü kendileri için ayırmıştır. Tek bir CPU olduğunu varsayarsak, işletim sistemi işlemlerden birini (diyelim ki A) çalıştırmayı seçerken, diğerleri (B ve C) hazır kuyruğunda çalışmayı beklemektedir. Zaman paylaşımı daha popüler hale geldikçe, işletim sisteminden yeni taleplerde bulunulduğunu tahmin edebilirsiniz. Özellikle, birden fazla programın aynı anda bellekte bulunmasına izin vermek, **korumayı (protection)** önemli bir konu haline getirir; bir işlemin bellekteki verileri okuyabilmesini veya Daha da kötüsü, başka bir sürecin belleğiniyazmak.



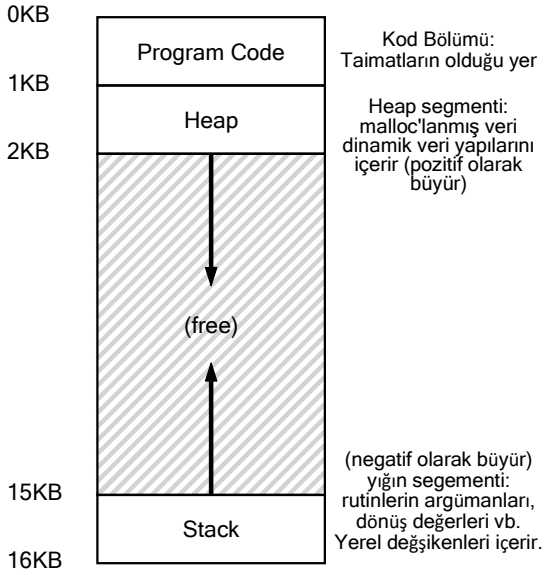
Şekil 13.2: **Üç Süreç (process): Hafıza paylaşımı**

13.3 Adres Alanları

Ancak, bu sinir bozucu kullanıcıları aklımızda tutmamız gerekiyor ve bunu yapmak için işletim sisteminin fiziksel belleğin **kullanımı kolay (easy to use)** bir soyutlamasını oluşturması gerekiyor. Bu soyutlamaya **adres uzayı (address space)** diyoruz ve çalışan programın sistemdeki bellek görünümüdür. Bu temel işletim sistemi bellek kısıtlamasını anlamak, belleğin nasıl sanallaştırıldığını anlamanın anahtarıdır. Bir sürecin adres alanı, çalışan programın tüm bellek durumunu içerir. Örneğin, programın **kodu(code)** (talimatlar) bellekte bir yerde yaşamak zorundadır ve bu nedenle adres uzayında yer alırlar. Program çalışırken, fonksiyon çağrı zincirinde nerede olduğunu takip etmek ve yerel değişkenleri tahsis etmek ve rutinlere parametreleri ve dönüş değerlerini aktarmak için bir **yığın (stack)** kullanır. Son olarak, **heap (yığın)** dinamik olarak tahsis edilen, kullanıcı tarafından yönetilen bellek için kullanılır, örneğin C'de `malloc()` veya C++ veya Java gibi nesne yönelimli bir dilde `new` çağrısından alabileceğiniz gibi. Elbette, burada başka şeyler de vardır (örneğin, statik olarak başlatılan değişkenler), ancak şimdilik sadece şu üç bileşeni varsayalım: kod, yığın ve yığın.

Şekil 13.3'teki örnekte, küçük bir adres alanımız (sadece 16KB)[1] vardır. Program kodu adres alanının en üstünde yer alır

[1] Genellikle bunun gibi küçük örnekler kullanırız çünkü (a) 32 bitlik bir adres alanını temsil etmek zahmetlidir ve (b) matematik daha zordur. Biz basit matematiği severiz.



Şekil 13.3: **Örnek Bir Adres Uzayı**

(bu örnekte 0'dan başlar ve adres alanının ilk 1K'sına yerleştirilir). Kod statiktir (ve bu nedenle belleğe yerleştirilmesi kolaydır), bu nedenle onu adres alanının en üstüne yerleştirebiliriz ve program çalışırken daha fazla alana ihtiyaç duymayacağını biliriz. Daha sonra, program çalışırken büyüyebilen (ve küçülebilen) adres alanının iki bölgesine sahibiz. Bunlar yığın (en üstte) ve yığın (en altta). Bunları bu şekilde yerleştiriyoruz çünkü her biri büyüyebilmek istiyor ve onları adres alanının zıt uçlarına koyarak böyle bir büyümeye izin verebiliyoruz: sadece zıt yönlerde büyümeleri gerekiyor. Böylece heap koddan hemen sonra başlar (1KB'de) ve aşağı doğru büyür (örneğin bir kullanıcı malloc() ile daha fazla bellek istediğinde); yığın 16KB'de başlar ve yukarı doğru büyür (örneğin bir kullanıcı bir işlem çağrısı yaptığında). Ancak, bu yığın ve yığın yerleşimi sadece bir kuraldır; isterseniz adres alanını farklı bir şekilde düzenleyebilirsiniz (daha sonra göreceğimiz gibi, bir adres alanında birden fazla **iş parçacığı (threads)** birlikte var olduğunda, adres alanını bu şekilde bölmenin güzel bir yolu artık işe yaramıyor, ne yazık ki).

Elbette, adres alanını tanımladığımızda, tanımladığımız şey işletim sisteminin çalışan programa sağladığı soyutlamadır. Program gerçekte 0 ila 16KB arasındaki fiziksel adreslerde bellekte değildir; bunun yerine bazı keyfi fiziksel adres(ler)e yüklenmiştir. Şekil 13.2'deki A, B ve C süreçlerini inceleyin; burada her sürecin belleğe nasıl farklı bir adresten yüklendiğini görebilirsiniz. Ve dolayısıyla sorun da budur:

KRİTİK NOKTA: BELLEK NASIL SANALLAŞTIRILIR
İşletim sistemi, tek bir fiziksel belleğin üzerinde birden fazla çalışan süreç (hepsi belleği paylaşan) için özel, potansiyel olarak büyük bir adres alanının bu soyutlamasını nasıl oluşturabilir?

İşletim sistemi bunu yaptığında, işletim sisteminin **belleği sanallaştırdığını (virtualizing memory)** söyleriz, çünkü çalışan program belleğe belirli bir adreste (örneğin 0) yüklendiğini ve potansiyel olarak çok büyük bir adres alanına (örneğin 32 bit veya 64 bit) sahip olduğunu düşünür; gerçek ise oldukça farklıdır. Örneğin, Şekil 13.2'deki A işlemi 0 adresinde (biz buna **sanal adres (virtual address**) diyeceğiz) bir yükleme yapmaya çalıştığında, işletim sistemi bir şekilde bazı donanım desteğiyle birlikte yüklemenin aslında 0 fiziksel adresine değil, 320KB fiziksel adresine (A'nın belleğe yüklendiği yer) gittiğinden emin olmak zorundadır. Bu, dünyadaki tüm modern bilgisayar sistemlerinin temelini oluşturan belleğin sanallaştırılmasının anahtarıdır.

13.4 Hedefler

Böylece bu notlar dizisinde işletim sisteminin görevine ulaşıyoruz: belleği sanallaştırmak. Ancak işletim sistemi yalnızca belleği sanallaştırmakla kalmayacak, bunu şık bir şekilde yapacaktır. İşletim sisteminin bunu yaptığından emin olmak için bize rehberlik edecek bazı hedeflere ihtiyacımız var. Bu hedefleri daha önce gördük (Giriş bölümünü düşünün) ve tekrar göreceğiz, ancak kesinlikle tekrar etmeye değer.

Bir sanal bellek (VM) sisteminin en önemli hedeflerinden biri **şeffaflıktır (transparency)**[2]. İşletim sistemi sanal belleği çalışan programa görünmeyecek şekilde uygulamalıdır. Bu nedenle, program belleğin sanallaştırıldığının farkında olmamalıdır; bunun yerine, program kendi özel fiziksel belleğine sahipmiş gibi davranır. Perde arkasında, işletim sistemi (ve donanım) belleği birçok farklı iş arasında çoklamak için tüm işi yapar ve dolayısıyla yanılsamayı uygular.

Sanal makinenin bir diğer hedefi de **verimliliktir (efficiency).** İşletim sistemi sanallaştırmayı hem zaman (yani programların çok daha yavaş çalışmasına neden olmamak) hem de alan (yani sanallaştırmayı desteklemek için gereken yapılar için çok fazla bellek kullanmamak) açısından mümkün olduğunca **verimli (efficient)** hale getirmeye çalışmalıdır. Zaman açısından verimli sanallaştırmanın uygulanmasında,

[2] Şeffaflığın bu kullanımı bazen kafa karıştırıcıdır; bazı öğrenciler "şeffaf olmanın" her şeyi açıkta tutmak, yani hükümetin nasıl olması gerektiği anlamına geldiğini düşünür. Burada ise tam tersi bir anlam ifade etmektedir: işletim sistemi tarafından sağlanan yanılsama uygulamalar tarafından görülememelidir. Dolayısıyla, yaygın kullanımda şeffaf bir sistem, Bilgi Edinme Özgürlüğü Yasası'nın öngördüğü şekilde taleplere yanıt veren değil, fark edilmesi zor olan bir sistemdir.

İPUCU: İZOLASYON İLKESİ

İzolasyon, güvenilir sistemler oluşturmada kilit bir ilkedir. İki varlık birbirinden uygun şekilde izole edilmişse, bu birinin diğerini etkilemeden arızalanabileceği anlamına gelir. İşletim sistemleri süreçleri birbirinden izole etmeye çalışır ve bu şekilde birinin diğerine zarar vermesini önler. Bellek izolasyonunu kullanarak işletim sistemi, çalışan programların altta yatan işletim sisteminin çalışmasını etkilememesini sağlar. Bazı modern işletim sistemleri, işletim sisteminin parçalarını işletim sisteminin diğer parçalarından ayırarak izolasyonu daha da ileri götürür. Bu tür **mikroçekirdekler (microkernels)** [BH70, R+89, S+03] bu nedenle tipik monolitik çekirdek tasarımlarından daha fazla güvenilirlik sağlayabilir.

işletim sistemi TLB'ler gibi donanım özellikleri de dahil olmak üzere donanım desteğine güvenmek zorunda kalacaktır (bunları ileride öğreneceğiz). Son olarak, üçüncü bir VM hedefi korumadır. İşletim sistemi, süreçleri birbirlerinden ve işletim sisteminin kendisini süreçlerden koruduğundan emin olmalıdır. Bir süreç bir yükleme, depolama ya da komut getirme işlemi gerçekleştirdiğinde, başka bir sürecin ya da işletim sisteminin kendisinin (yani kendi adres alanı dışındaki herhangi bir şeyin) bellek içeriğine erişememeli ya da bu içeriği herhangi bir şekilde etkileyememelidir. Böylece koruma, süreçler arasında **izolasyon (isolation)** özelliğini sunmamızı sağlar; her bir süreç, diğer hatalı ve hatta kötü niyetli süreçlerin tahribatından korunarak kendi izole konusunda çalışmalıdır.

Sonraki bölümlerde, donanım ve işletim sistemi desteği de dahil olmak üzere belleği sanallaştırmak için gereken temel **mekanizmalara (mechanisms)** odaklanacağız. Ayrıca, boş alanın nasıl yönetileceği ve alanınız azaldığında hangi sayfaların bellekten atılacağı gibi işletim sistemlerinde karşılaşacağınız daha ilgili bazı **politikaları (policies)** da inceleyeceğiz. Bunu yaparken, modern bir işletim sisteminin nasıl çalıştığına dair anlayışınızı geliştireceğiz. Sanal bellek sistemi gerçekten çalışıyor[3].

[3] Ya da sizi kursu bırakmaya ikna edeceğiz. Ama bekleyin; eğer VM'den geçmeyi başarırsanız, muhtemelen sonuna kadar gideceksiniz!

13.5 Özet

Önemli bir işletim sistemi alt sisteminin tanıtıldığını gördük: sanal bellek. Sanal bellek sistemi programlara büyük, seyrek, özel bir adres alanı yanılsaması sağlamaktan sorumludur ve tüm talimatlarını ve verilerini burada tutar. İşletim sistemi, bazı ciddi donanım yardımlarıyla, bu sanal bellek referanslarının her birini alacak ve istenen bilgiyi almak için fiziksel belleğe sunulabilecek fiziksel adreslere dönüştürecektir. İşletim sistemi bunu aynı anda birçok işlem için yapacak ve programları birbirinden koruduğu gibi işletim sistemini de koruyacaktır. Tüm bu yaklaşımın çalışması için çok sayıda mekanizmanın (çok sayıda düşük seviyeli makine) yanı sıra bazı kritik politikalar da gerekmektedir; önce kritik mekanizmaları açıklayarak aşağıdan yukarıya doğru başlayacağız. Ve böylece devam ediyoruz!

YANINDA: GÖRDÜĞÜNÜZ HER ADRES SANALDIR.

Hiç işaretçi yazdıran bir C programı yazdınız mı? Gördüğünüz değer (genellikle onaltılık olarak yazdırılan büyük bir sayı) sanal bir adrestir. Programınızın kodunun nerede bulunduğunu hiç merak ettiniz mi? Bunu da yazdırabilirsiniz ve evet, eğer yazdırabiliyorsanız, bu da bir sanal adrestir. Aslında, kullanıcı düzeyinde bir programın programcısı olarak görebileceğiniz her adres sanal bir adrestir. Bu talimatların ve veri değerlerinin makinenin fiziksel belleğinin neresinde olduğunu sadece işletim sistemi, belleği sanallaştırma teknikleri sayesinde bilir. Bu yüzden asla unutmayın: bir programda bir adres yazdırırsanız, bu sanal bir adrestir, bellekte işlerin nasıl düzenlendiğine dair bir yanılsamadır; gerçek gerçeği yalnızca işletim sistemi (ve donanım) bilir.

İşte `main()` rutininin (kodun bulunduğu yer) konumlarını, `malloc()`'tan dönen heap'e ayrılmış bir değerin değerini ve yığındaki bir tamsayının konumunu yazdıran küçük bir program (va.c):

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(int argc, char *argv[]) {
4   printf("location of code : %p\n", main);
5   printf("location of heap : %p\n", malloc(100e6));
6   int x = 3;
7   printf("location of stack: %p\n", &x);
8   return x;
9 }
```

64 bit Mac üzerinde çalıştırıldığında aşağıdaki çıktıyı alıyoruz:

```
location of code  : 0x1095afe50
location of heap  : 0x1096008c0
location of stack: 0x7fff691aea64
```

Buradan, kodun adres alanında önce geldiğini, sonra yığının geldiğini ve yığının bu büyük sanal alanın diğer ucunda olduğunu görebilirsiniz. Tüm bu adresler sanaldır ve değerleri gerçek fiziksel konumlarından almak için işletim sistemi ve donanım tarafından çevrilecektir.

References

[BH70] "The Nucleus of a Multiprogramming System" by Per Brinch Hansen. Communications of the ACM, 13:4, April 1970. *The first paper to suggest that the OS, or kernel, should be a minimal and flexible substrate for building customized operating systems; this theme is revisited throughout OS research history.*

[CV65] "Introduction and Overview of the Multics System" by F. J. Corbato, V. A. Vyssotsky. Fall Joint Computer Conference, 1965. *A great early Multics paper. Here is the great quote about time sharing: "The impetus for time-sharing first arose from professional programmers because of their constant frustration in debugging programs at batch processing installations. Thus, the original goal was to time-share computers to allow simultaneous access by several persons while giving to each of them the illusion of having the whole machine at his disposal."*

[DV66] "Programming Semantics for Multiprogrammed Computations" by Jack B. Dennis, Earl C. Van Horn. Communications of the ACM, Volume 9, Number 3, March 1966. *An early paper (but not the first) on multiprogramming.*

[L60] "Man-Computer Symbiosis" by J. C. R. Licklider. IRE Transactions on Human Factors in Electronics, HFE-1:1, March 1960. *A funky paper about how computers and people are going to enter into a symbiotic age; clearly well ahead of its time but a fascinating read nonetheless.*

[M62] "Time-Sharing Computer Systems" by J. McCarthy. Management and the Computer of the Future, MIT Press, Cambridge, MA, 1962. *Probably McCarthy's earliest recorded paper on time sharing. In another paper [M83], he claims to have been thinking of the idea since 1957. McCarthy left the systems area and went on to become a giant in Artificial Intelligence at Stanford, including the creation of the LISP programming language. See McCarthy's home page for more info:* `http://www-formal.stanford.edu/jmc/`

[M+63] "A Time-Sharing Debugging System for a Small Computer" by J. McCarthy, S. Boilen, E. Fredkin, J. C. R. Licklider. AFIPS '63 (Spring), New York, NY, May 1963. *A great early example of a system that swapped program memory to the "drum" when the program wasn't running, and then back into "core" memory when it was about to be run.*

[M83] "Reminiscences on the History of Time Sharing" by John McCarthy. 1983. Available: http://www-formal.stanford.edu/jmc/history/timesharing/timesharing.html. *A terrific historical note on where the idea of time-sharing might have come from including some doubts towards those who cite Strachey's work [S59] as the pioneering work in this area.*

[NS07] "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation" by N. Nethercote, J. Seward. PLDI 2007, San Diego, California, June 2007. *Valgrind is a lifesaver of a program for those who use unsafe languages like C. Read this paper to learn about its very cool binary instrumentation techniques – it's really quite impressive.*

[R+89] "Mach: A System Software kernel" by R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, M. Jones. COMPCON '89, February 1989. *Although not the first project on microkernels per se, the Mach project at CMU was well-known and influential; it still lives today deep in the bowels of Mac OS X.*

[S59] "Time Sharing in Large Fast Computers" by C. Strachey. Proceedings of the International Conference on Information Processing, UNESCO, June 1959. *One of the earliest references on time sharing.*

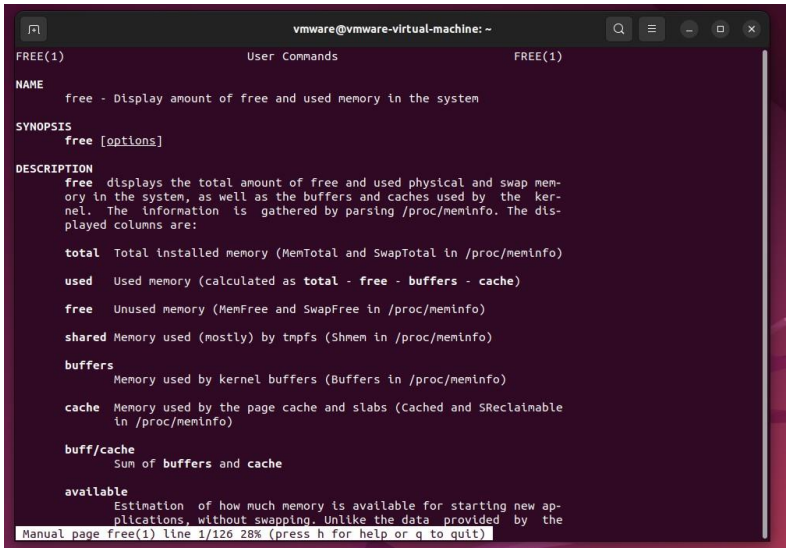
[S+03] "Improving the Reliability of Commodity Operating Systems" by M. M. Swift, B. N. Bershad, H. M. Levy. SOSP '03. *The first paper to show how microkernel-like thinking can improve operating system reliability.*

ÖDEV (Code)

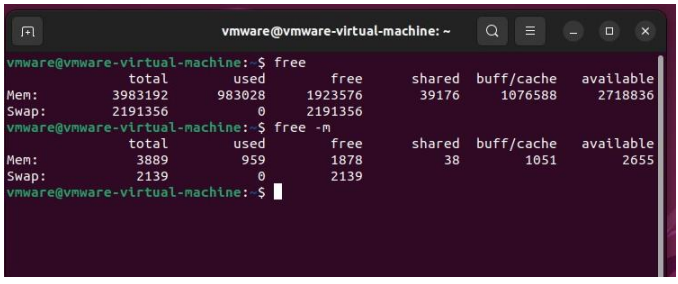
Bu ödevde, Linux tabanlı sistemlerde sanal bellek kullanımını incelemek için sadece birkaç yararlı araç hakkında bilgi edineceğiz. Bu sadece nelerin mümkün olduğuna dair kısa bir ipucu olacak; gerçekten bir uzman olmak için kendi başınıza daha derine dalmanız gerekecek (her zaman olduğu gibi!).

Sorular

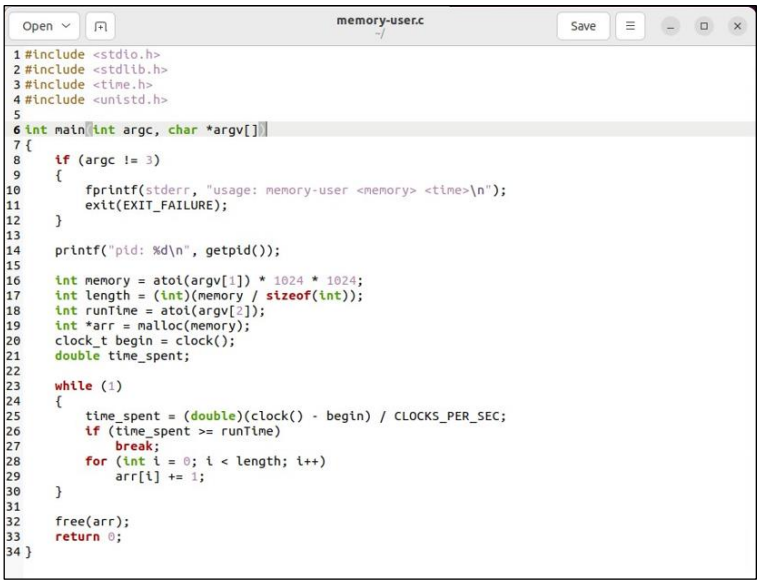
1. Kontrol etmeniz gereken ilk Linux aracı çok basit bir araç olan free'dir. Öncelikle, man free yazın ve tüm kılavuz sayfasını okuyun; kısa, endişelenmeyin!



2. Şimdi, belki yararlı olabilecek bazı argümanları kullanarak (örneğin, bellek toplamlarını megabayt cinsinden görüntülemek için -m) serbestçe çalıştırın. Sisteminizde ne kadar bellek var? Ne kadarı boş? Bu rakamlar sezgilerinizle uyuşuyor mu?
4 Gb bellek var. Yaklaşık 1gb kullanılıyor geri kalanı boş

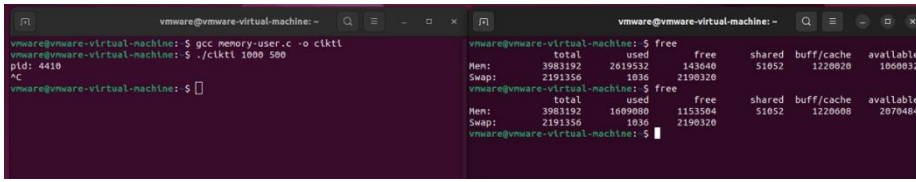


3. Daha sonra, memory-user.c adında belirli miktarda bellek kullanan küçük bir program oluşturun. Bu program bir komut satırı argümanı almalıdır: kullanacağı bellek megabayt sayısı. Çalıştırıldığında, bir dizi tahsis etmeli ve her girdiye dokunarak dizi boyunca sürekli akış yapmalıdır. Program bunu süresiz olarak veya belki de komut satırında belirtilen belirli bir süre boyunca yapmalıdır.
   İki adet komut satırı argümanı alan bir program oluşturuldu. Bunlardan biri kullanılacak bellek boyutu (mb cinsinden), ikincisi ise runtime yani çalışma süresi (sn).

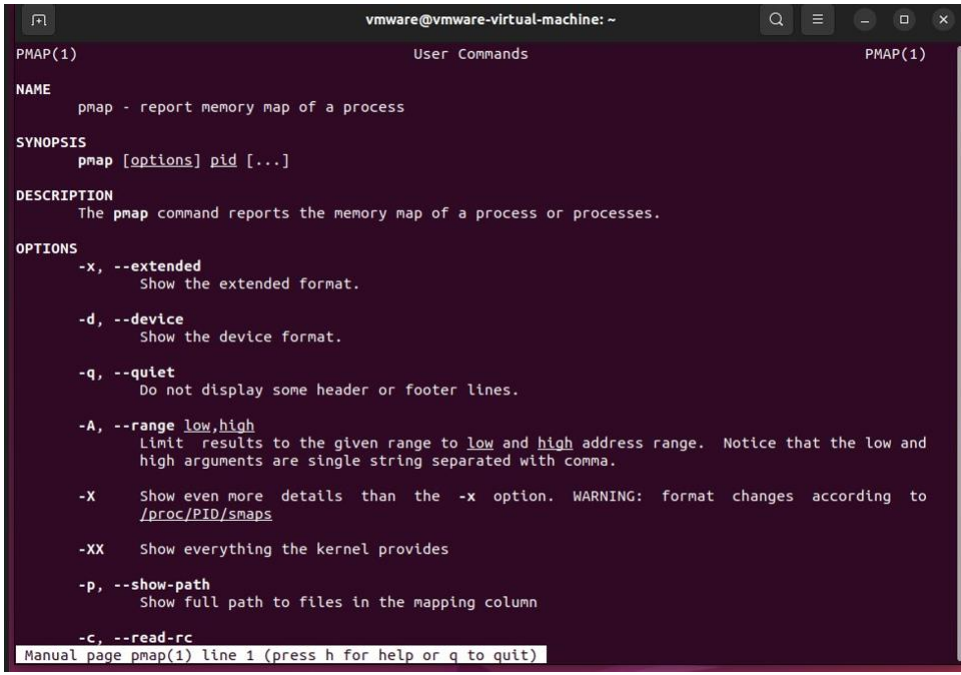



```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        fprintf(stderr, "usage: memory-user <memory> <time>\n");
        exit(EXIT_FAILURE);
    }

    printf("pid: %d\n", getpid());

    int memory = atoi(argv[1]) * 1024 * 1024;
    int length = (int)(memory / sizeof(int));
    int runTime = atoi(argv[2]);
    int *arr = malloc(memory);
    clock_t begin = clock();
    double time_spent;

    while (1)
    {
        time_spent = (double)(clock() - begin) / CLOCKS_PER_SEC;
        if (time_spent >= runTime)
            break;
        for (int i = 0; i < length; i++)
            arr[i] += 1;
    }

    free(arr);
    return 0;
}
```

4. Şimdi, bellek kullanıcısı programınızı çalıştırırken, aynı zamanda (farklı bir terminal penceresinde, ancak aynı makinede) serbest aracı çalıştırın. Programınız çalışırken bellek kullanım toplamları nasıl değişiyor? Peki ya bellek-kullanıcı programını öldürdüğünüzde? Rakamlar beklentilerinizle uyuşuyor mu? Bunu farklı miktarlarda bellek kullanımı için deneyin. Gerçekten büyük miktarlarda bellek kullandığınızda ne olur?
   Beklentilerimizle uyuştu. Programda komut satırı argümanı verdiğimiz 1 mblik bellek program çalıştığı sürede kullanıldı. Used bölümünde belleği görüntülemiş olduk. Programı kapattığımızda da belleğe tekrardan belleğin kullanılabilir olduğunu görmüş olduk.

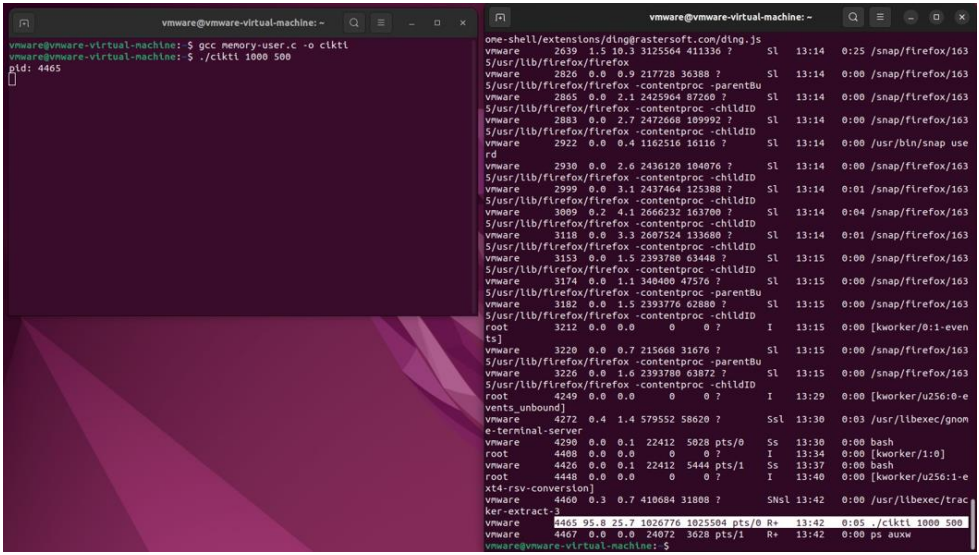


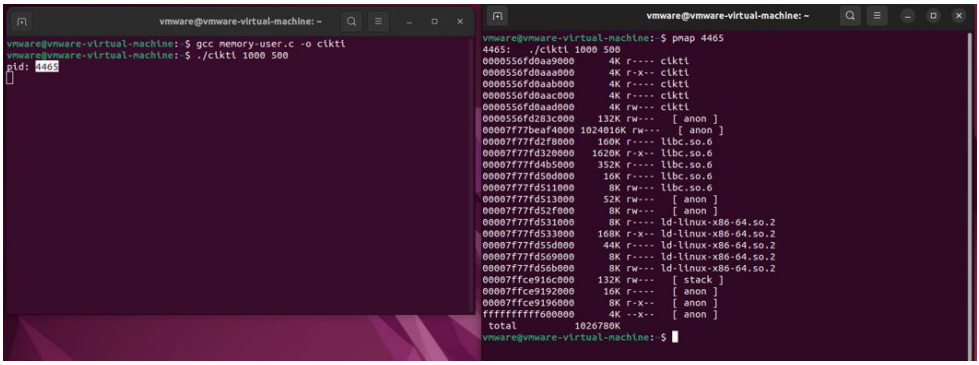
5. Şimdi pmap olarak bilinen bir aracı daha deneyelim. Biraz zaman ayırın ve pmap kılavuz sayfasını ayrıntılı olarak okuyun.



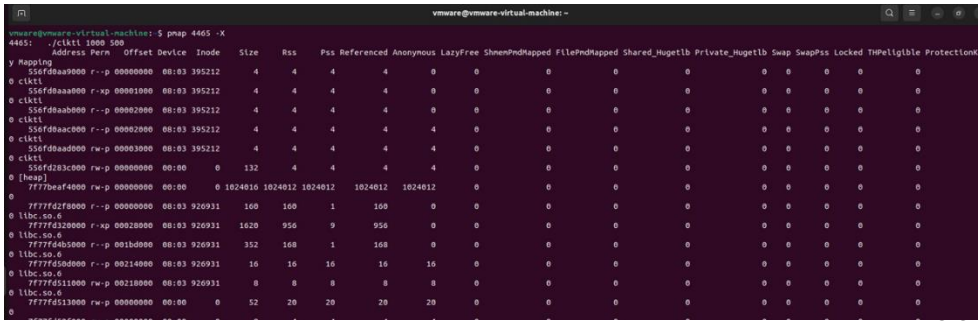
6. pmap'i kullanmak için, kullandığınız sürecin süreç kimliğini bilmeniz gerekir. ilgilendiğiniz. Bu nedenle, önce tüm süreçlerin bir listesini görmek için ps auxw komutunu çalıştırın; ardından, tarayıcı gibi ilginç bir tanesini seçin. Bu durumda bellek kullanıcısı programınızı da kullanabilirsiniz (hatta bu programın getpid()'i çağırmasını ve size kolaylık sağlamak için PID'sini yazdırmasını sağlayabilirsiniz).

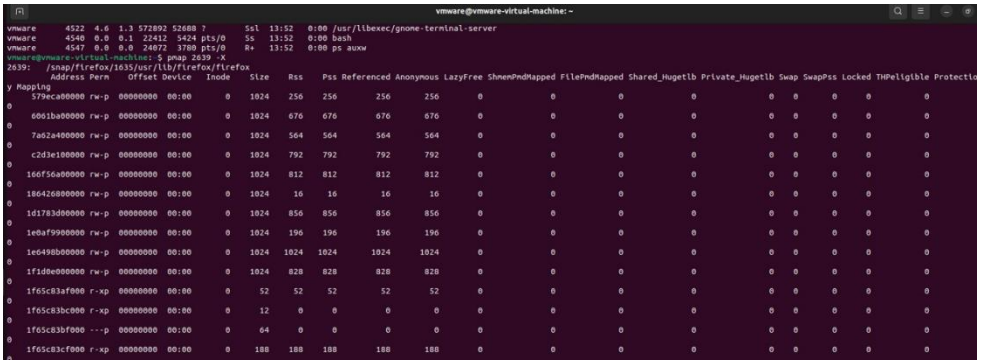
Programın içerisinde getpid() komutunu çalıştırıp, programın pidsini elde ettim.





7. Şimdi çeşitli bayraklar kullanarak bu süreçlerden bazıları üzerinde pmap çalıştırın (örneğin -X) tuşuna basarak süreçle ilgili birçok ayrıntıyı görebilirsiniz. Ne görüyorsunuz? Basit kod/yığın/yığın anlayışımızın aksine, modern bir adres alanını kaç farklı varlık oluşturur?





8. Son olarak, farklı miktarlarda kullanılan bellek ile bellek-kullanıcı programınız üzerinde pmap çalıştıralım. Burada ne görüyorsunuz? Pmap çıktısı beklentilerinizle uyuşuyor mu?
   Çıktı beklentileri karşılıyor. Verdiğimiz komut satırı argümanına göre kullandığı bellek miktarı değişiyor.