13

Soyutlama: Adres Alanları

İlk günlerde, bilgisayar sistemleri kurmak kolaydı. Neden, diye soruyorsunuz? Çünkü kullanıcılar fazla bir şey beklemiyordu. Bu cüretkar kullanıcılar, "kullanım kolaylığı", "yüksek performans", "güvenilirlik" vb. beklentiler, gerçekten tüm bu baş ağrılarına yol açtı. Bir dahaki sefere bunlardan biriyle tanıştığında bilgisayar kullanıcıları, neden oldukları tüm sorunlar için onlara teşekkür eder.

13.1 Eski Sistemler

Bellek açısından bakıldığında, ilk makineler kullanıcılara çok fazla soyutlama sağlamadı. Temel olarak, makinenin fiziksel belleği Şekil 13.1'de (sayfa 2) gördüğünüz gibi görünüyordu.

İşletim sistemi, bellekte bulunan (bu örnekte fiziksel adres 0'dan başlayan) bir dizi rutindi (aslında bir kitaplık), ve şu anda fiziksel bellekte (bu örnekte 64k fiziksel adresten başlayan) oturan ve belleğin geri kalanını kullanan çalışan bir program (bir işlem) olacaktır. Burada çok az yanılsama vardı ve kullanıcı işletim sisteminden pek 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 İnsanlar makineleri daha etkin bir şekilde kullanmaya başladı. Böylece çoklu programlama **multiprogramming (çoklu programlama)** çağı doğdu [DV66], burada birden fazla süreç belirli bir zamanda çalışmaya hazırdı ve örneğin bir I/O gerçekleştirmeye karar verildiğinde işletim sistemi bunlar arasında geçiş yapıyordu. Bunu yapmak işlemcide ki etkin kullanımı(**utilization**) arttırdı. **Verimlilikte** (**efficiency)** ki bu tür artışlar, her bir makinenin binlerce dolara mal olduğu günlerde çok önemliydi (Ve siz MAC' inizin çok pahalı olduğunu düşünüyordunuz!).

Ancak çok geçmeden insanlar makinelerden daha fazlasını talep etmeye başladı ve **time sharing(zaman paylaşımı)** çağı doğdu. [S59, L60, M62, M83]. Özellikle de uzun (ve dolayısıyla etkisiz) program hata ayıklama döngülerinden bıkmış olan profesyonel programcılar [CV65] başta olmak üzere pek çok kişi toplu hesaplamanın sınırlamalarının farkına vardı.

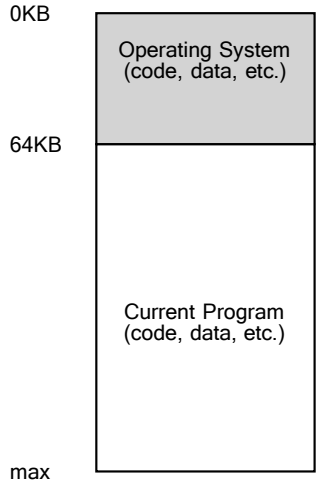

Figure 13.1: **İşletim Sistemleri: İlk Günler**

Birçok kullanıcı aynı anda bir makineyi kullanıyor olabileceğinden ve her biri o anda yürüttükleri görevlerden zamanında yanıt beklediğinden (ya da umduğundan) **etkileşim (interactivity)** kavramı önem kazandı. Zaman paylaşımını uygulamanın bir yolu, bir süreci kısa bir süre çalıştırmak, ona tüm belleğe tam erişim vermek (Şekil 13.1), sonra onu durdurmak, tüm durumunu bir tür diske kaydetmek (tüm fiziksel bellek dahil), başka bir sürecin durumunu yüklemek, onu bir süre çalıştırmak ve böylece makinenin bir tür kaba paylaşımını uygulamak olabilir [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, sayfa 3'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, muhtemelen işletim sistemine yeni talepler geldiğini tahmin edebilirsiniz. Özellikle, birden fazla programın aynı anda bellekte bulunmasına izin vermek, **korumayı (protec-tion)** önemli bir konu haline getirir; bir sürecin bellekteki verileri okuyabilmesini veya daha da kötüsü, başka bir sürecin belleğini yazmak.

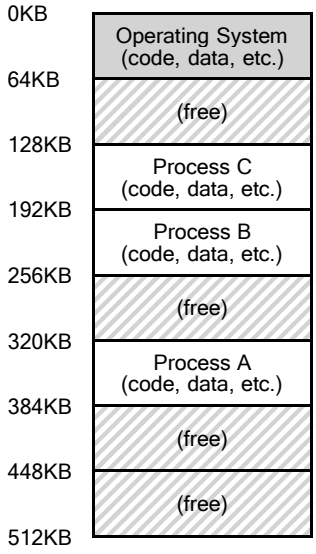

Figure 13.2: **Üç Süreç: Bellek Paylaşımı**

13.3 Adres ALanı

Ancak, bu sinir bozucu kullanıcıları aklımızda tutmamız gerekiyor ve bunu yapmak için işletim sisteminin fiziksel belleğin **kullanımı kolay (easy yo 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 (talimatlar) bellekte bir yerde yaşamak zondruadır ve bu nedenle adress 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 kullanır . Son olarak **heap (yığın),** C'de malloc() veya C++ veya Java gibi nesne yönelimli bir dilde new çağrısından alabileceğiniz gibi dinamik olarak tahsis edilen, kullanıcı tarafından yönetilen bellek için kullanılır. Elbette, orada başka şeyler de var (örneğin, statik olarak başlatılan değişkenler), ancak şimdilik sadece şu üç bileşeni varsayalım; kod, yığın ve yığın. Şekil 13.3'teki (sayfa 4) örnekte, küçük bir adres alanımız vardır (sadece 16KB)1. Program kodu adres alanının en üstünde yer alır (bu örnekte 0'dan başlar ve adres alanının ilk 1K'sına yerleştirilir).

11Genellikle bunun gibi küçük örnekler kullanırız çünkü (a) 32 bitlik bir adres alanını temsil etmek zahmetlidir ve (b) matematik daha zordur. Basit matematiği severiz.

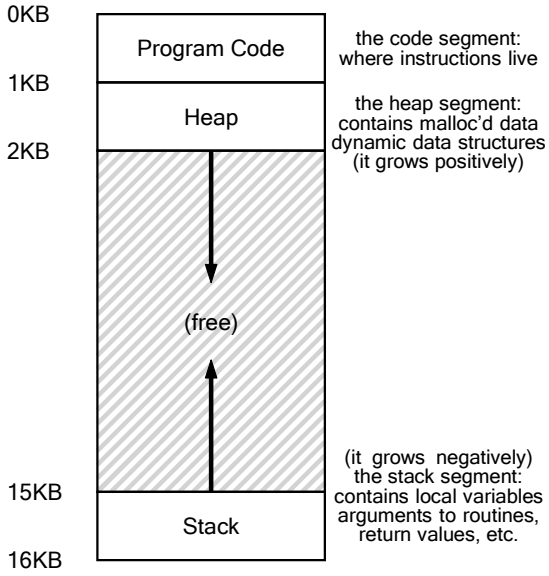

Figure 13.3: **Örnek Bir Adres Alanı**

Kod statiktir (ve bu nedenle belleğe yerleştirilmesi kolaydır), bu nedenle onu adres alanının en üstüne yerleştirebilir ve program çalıştıkça 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 bu tür bir büyümeye izin verebiliriz: sadece zıt yönlerde büyümeleri gerekir. Böylece heap koddan hemen sonra başlar (1KB'de) ve aşağı doğru büyür (örneğin bir kullanıcı malloc() ile daha fazla bellek istediğinde); stack 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 çalışmıyor, ne yazık ki).

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

THE CRUX: HOW TO VIRTUALIZE MEMORY

İşletim sistemi, tek bir fiziksel belleğin üzerinde birden fazla çalışan süreç 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 ad- dress'de (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 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 her modern bilgisayar sisteminin 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.

VM'nin 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 hale getirmeye çalışmalıdır. Zaman açısından verimli sanallaştırmanın uygulanmasında, işletim sistemi TLB'ler gibi donanım özellikleri de dahil olmak üzere donanım desteğine güvenmek zorunda kalacaktır (zamanı gelince öğreneceğiz).

Son VM hedefi ise **korumadır (protection)**. İşletim sistemi, süreçleri birbirlerinden ve işletim sisteminin kendisini süreçlerden **koruduğundan** emin olmalıdır.

[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.

TIP: THE PRINCIPLE OF ISOLATION
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 **mikro çekirdekler (microkernels)** [BH70, R+89, S+03] bu nedenle tipik monolitik çekirdek tasarımlarından daha fazla güvenilirlik sağlayabilir.

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 özelliğini sunmamızı sağlar; her bir süreç, diğer hatalı ve hatta kötü niyetli süreçlerin tahribatından korunarak kendi izole koonusunda çalışmalıdır.

Sonraki bölümlerde, donanım ve işletim sistemi desteği de dahil olmak üzere, belleği sanallaştırmak için gereken temel **mekanizmaları (nisms)** incelemeye 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 de, modern bir modernitenin nasıl işlediğine dair anlayışınızı geliştireceğiz.

13.5 Özet

Önemli bir işletim sistemi alt sisteminin tanıtıldığını gördük: sanal bellek. Sanal bellek sistemi, tüm talimatlarını ve verilerini burada tutan programlara büyük, seyrek, özel bir adres alanı yanılsaması sağlamaktan sorumludur. 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; öncekritikmekanizmazları açıklayarak aşağıdan yukarıya doğru başlayacağız. Ve böylece devam ediyoruz!

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

ASIDE: EVERY ADDRESS YOU SEE IS VIRTUAL

Ever write a C program that prints out a pointer? The value you see (some large number, often printed in hexadecimal), is a **virtual address**. Ever wonder where the code of your program is found? You can print that out too, and yes, if you can print it, it also is a virtual address. In fact, any address you can see as a programmer of a user-level program is a virtual address. It's only the OS, through its tricky techniques of virtualizing memory, that knows where in the physical memory of the machine these instructions and data values lie. So never forget: if you print out an address in a program, it's a virtual one, an illusion of how things are laid out in memory; only the OS (and the hardware) knows the real truth.

Here's a little program (`va.c`) that prints out the locations of the `main()` routine (where code lives), the value of a heap-allocated value returned from `malloc()`, and the location of an integer on the stack:

```
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 }
```

When run on a 64-bit Mac, we get the following output:

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

From this, you can see that code comes first in the address space, then the heap, and the stack is all the way at the other end of this large virtual space. All of these addresses are virtual, and will be translated by the OS and hardware in order to fetch values from their true physical locations.

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

14

1. Kontrol etmeniz gereken ilk şey Linux aracı, ücretsiz ve çok basit bir araçtır. Önce 'man free' yazın ve tüm kılavuz sayfasını okuyun. Kısa oldu, merak etmeyin!

2. Şimdi yararlı olabilecek argümanlardan bazılarını kullanaral özgürce koşun. (e.g., `-m`, to display memory totals in megabytes). Sisteminizde ne kadar bellek var? Ne kadarı özgür? Bu sayılar sizinkiyle eşleşiyor mu?
   İşletim sistemi sanal bir bellekte çalıştığı için daha az. Program durduğunda belleğin azaldığı görülür.

```
[shubham@localhost ~]$ free -m
              total        used        free      shared  buff/cache   available
Mem:           1839         883         758           3         197         771
Swap:          1227         169        1058
[shubham@localhost ~]$ free -m
              total        used        free      shared  buff/cache   available
Mem:           1839         648         984           3         206        1005
Swap:          1227         167        1060
[shubham@localhost ~]$
```

3. Daha sonra, `memory-user.c` adlı, belirli bir 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.

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

    n=atoi(argv[1]);
    m=1024*n;
    if(argc==2){
        a=(char*)malloc(n*1024);
        printf("Ayrılan belek ");
    }
    if(argc==3){
        m=atoi(argv[3]);
    }
    /*
     Program bunu süresiz olarak veya belki de komut satırında belirtilen belirli bir süre boyunca yapmalıdır.
    */
    printf("diziye erişim...");
    for(i=0;i<m;i++)
            printf("%c\t",a[i]);
```

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?

   watch -n 1 free -m kullanarak belleği takip edip Memory-user programı çalıştırıldığında, argüman olarak verilenz MB miktarının free tarafından gösterilen istatistiklere eklendiğini görürüz. Program öldürüldüğünde, kullanılan bellek argüman olarak verdiğiniz MB miktarı kadar azalır. Büyük miktarda bellek girildiğinde takasın dolduğunu görürüz.

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 id'sini (process ID)** 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() işlevini çağırmasını ve size kolaylık sağlamak için PID'sini yazdırmasını sağlayabilirsiniz).

7. Şimdi çeşitli pmağler kullanarak bu işlemlerden bazılarında çalıştırın(Örneğin -X tuşuna basarak bir çok ayrıntıyı görebilirsiniz).Ne görüyorsunuz? Basit kod / yığın/ heap anlayışımızın aksine modern adres anlayışını kaç farklı varlık oluşturur ?

8. Son olarak, farklı miktarlarda kullanılan bellek ile bellek kullanıcı programınız üzerinde pmap'i çalıştıralım. Burada ne görüyorsunuz? Pmap çıktısı beklentilerinizle uyuşuyor mu?