13

Soyutlama: Adres Uzayları

İlk günlerde, bilgisayar sistemleri kurmak kolaydı. Neden, diye soruyorsunuz? Çünkü kullanıcılar fazla bir şey beklemiyordu. Tüm bu baş ağrılarına gerçekten yol açan, "kullanım kolaylığı", "yüksek performans", "güvenilirlik" vb. beklentileriyle cesur kullanıcılardır. Bir dahaki sefere bu bilgisayar kullanıcılarından biriyle tanıştığınızda, neden oldukları tüm sorunlar için onlara teşekkür edin.

13.1 İlk Sistemler

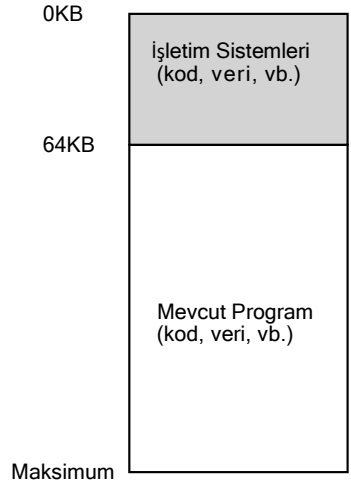
Bellek açısından bakıldığında, ilk makineler kullanıcılara pek bir soyutlama sağlamıyordu. Temel olarak, makinenin fiziksel belleği Şekil 13.1'de (sayfa 2) gördüğünüz gibi görünüyordu.

İşletim sistemi, bellekte bulunan (bu örnekte fiziksel adres 0'dan başlayarak) bir dizi rutindi (bir kütüphane, gerçekten) ve şu anda fiziksel bellekte bulunan (bu örnekte fiziksel adres 64k'dan başlayarak) ve belleğin geri kalanını kullanan bir çalıştırma programı (bir işlem) olacaktı. 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 etkili bir şekilde paylaşmaya başladılar. Böylece, birden fazla işlemin belirli bir zamanda çalışmaya hazır olduğu ve örneğin bir I/O(Giriş/Çıkış) gerçekleştirmeye karar verildiğinde işletim sisteminin bunlar arasında geçiş yaptığı **çoklu programlama (multiprogramming)** çağı doğdu [DV66]. Bunu yapmak, CPU'nun etkin **kullanımını(utilization)** artırdı. **Verimlilikteki (efficiency)** bu tür artışlar, her makinenin yüz binlerce hatta milyonlarca dolara mal olduğu (ve Mac'inizin pahalı olduğunu düşündünüz!) günlerde özellikle önemliydi.

Ancak çok geçmeden insanlar daha fazla makine talep etmeye başladı ve **zaman paylaşımı (time sharing)** çağı doğdu [S59, L60, M62, M83]. Spesifik olarak, birçoğu, özellikle uzun (ve dolayısıyla etkisiz) program hata ayıklama döngülerinden bıkmış olan programcıların kendileri [CV65] üzerinde toplu hesaplamanın



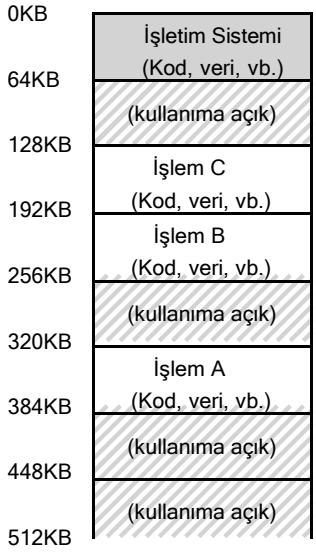
Şekil 13.1: **İşletim Sistemleri: İlk Zamanlar**

sınırlamalarını fark etti. **Etkileşim (interactivity)** kavramı önemli hale geldi çünkü birçok kullanıcı aynı anda bir makineyi kullanıyor olabilir, her biri o anda yürütmekte oldukları görevlerden zamanında yanıt bekliyor (veya umuyor). Zaman paylaşımını uygulamanın bir yolu, bir işlemi kısa bir süre için çalıştırın, ona tüm belleğe tam erişim verin (Şekil 13.1), sonra durdurun, tüm durumunu bir tür diske kaydedin (tüm fiziksel bellek dahil), başka bir işlemin durumunu yükleyin, bir süre çalıştırın ve Makinenin bir tür ham paylaşımı gerçekleştirin [M+63].

Ne yazık ki, bu yaklaşımın büyük bir sorunu var: özellikle hafıza büyüdükçe çok yavaş. Kayıt düzeyi durumunu (PC, genel amaçlı kayıtlar, vb.) kaydetmek ve geri yüklemek nispeten hızlı olsa da belleğin tüm içeriğini diske kaydetmek acımasızca performans göstermez. Bu nedenle, yapmayı tercih ettiğimiz şey, işlemler arasında geçiş yaparken işlemleri bellekte bırakarak işletim sisteminin zaman paylaşımını verimli bir şekilde uygulamasına izin vermektir (Şekil 13.2, sayfa 3'te gösterildiği gibi).

Diyagramda, üç işlem vardır (A, B ve C) ve bunların her biri, kendileri için oyulmuş 512 KB'lık fiziksel belleğin küçük bir bölümüne sahiptir. Tek bir CPU varsayıldığında, işletim sistemi süreçlerden birini (örneğin A) çalıştırmayı seçer, diğerleri ise (B ve C) hazır kuyruğunda çalışmayı bekler. Zaman paylaşımı daha popüler hale geldikçe, muhtemelen işletim sistemine yeni talepler getirildiğini tahmin edebilirsiniz.

Özellikle birden çok programın aynı anda bellekte bulunmasına izin verilmesi, **korumayı(protection)** önemli bir sorun haline getirir; okuyabilmek için bir süreç istemiyorsunuz veya daha da kötüsü, başka bir işlemin belleğini yazın.



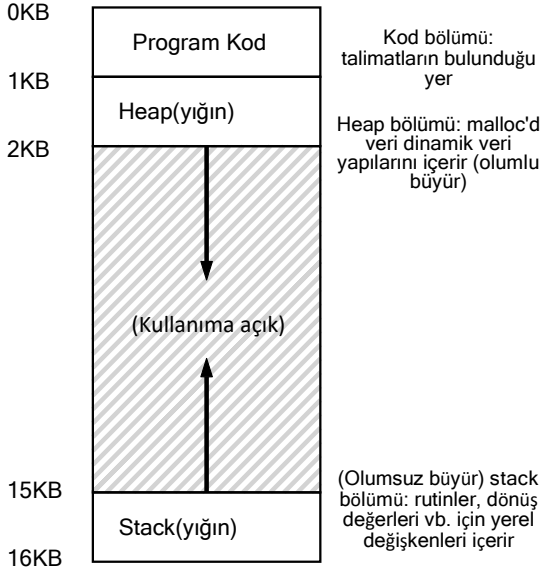
Şekil 13.2: **Üç İşlem: Bellek Paylaşımı**

13.3 Adres Uzayı

Bununla birlikte, bu sinir bozucu kullanıcıları akılda tutmalıyız ve bunu yapmak, işletim sisteminin fiziksel belleğin **kullanımı kolay (easy to use)** bir soyutlamasını oluşturmasını gerektirir.Bu soyutlamaya **adres uzayı (address spaces)** diyoruz ve çalışan programın sistemdeki bellek görünümüdür. Belleğin bu temel işletim sistemi soyutlamasını anlamak, belleğin nasıl sanallaştırıldığını anlamanın anahtarıdır. Bir işlemin adres uzayı, çalışan programın tüm bellek durumunu içerir. Örneğin, programın **kodu (code)** (talimatlar) bellekte bir yerde bulunmak zorundadır ve bu nedenle adres uzayında bulunurlar. Program çalışırken, işlev çağrı zincirinde nerede olduğunu takip etmek, ayrıca yerel değişkenleri ayırmak, parametreleri iletmek, yordamlara ve yordamlardan değerler döndürmek için **yığın(stack)** kullanılır. Son olarak, **yığın(heap)** C'deki bir `malloc()` çağrısından veya C++ ya da Java gibi nesne yönelimli bir dilde yeni bir çağrıdan alabileceğiniz gibi, dinamik olarak ayrılmış, kullanıcı tarafından yönetilen bellek için kullanılır Tabii ki, orada başka şeyler de var (örneğin, statik olarak başlatılmış değişkenler) ama şimdilik sadece bu üç bileşeni varsayalım: kod, stack(yığın) ve heap(yığın).

Şekil 13.3'teki örnekte (sayfa 4), çok küçük bir adres uzayımız var (yalnızca 16KB)[1] . Program kodu, adres uzayının en üstünde yer alır

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



Şekil 13.3: **Örnek Adres Uzayı**

(Bu örnekte 0'dan başlayarak ve adres uzayının ilk 1K'lık bölümüne yerleştirilmiştir). Kod statiktir (ve bu nedenle belleğe yerleştirilmesi kolaydır), bu nedenle onu adres uzayının en üstüne yerleştirebilir ve program çalışırken daha fazla alana ihtiyaç duymayacağını bilebiliriz.

Daha sonra, program çalışırken büyüyebilecek (ve küçülebilecek) adres uzayının iki bölgesine sahibiz. Bunlar heap(yığın) (üstte) ve stack(yığın) (altta). Onları bu şekilde yerleştiriyoruz çünkü her biri büyüyebilmek istiyor ve onları adres uzayının zıt uçlarına koyarak böyle bir büyümeye izin verebiliriz: sadece zıt yönlerde büyümeleri gerekiyor. Böylece yığın, koddan hemen sonra başlar (1 KB'de) ve aşağı doğru büyür (örneğin, bir kullanıcı `malloc()` aracılığıyla daha fazla bellek istediğinde); yığın 16 KB'de başlar ve yukarı doğru büyür (örneğin, bir kullanıcı bir prosedür çağrısı yaptığında). Ancak, yığının ve öbeğin bu yerleşimi yalnızca bir kuraldır; isterseniz adres uzayını farklı bir şekilde düzenleyebilirsiniz (daha sonra göreceğimiz gibi, bir adres uzayında birden fazla **iş parçacığı(threads)** bir arada bulunduğunda, adres uzayını bu şekilde bölmenin artık iyi bir yolu yok, ne yazık ki).

Elbette, adres alanını tanımladığımızda, tanımladığımız şey, işletim sisteminin çalışan programa sağladığı **soyutlamadır(abstraction)**. Program, 0 ile 16KB arasındaki fiziksel adreslerde gerçekten bellekte değildir; bunun yerine bazı rasgele fiziksel adreslere yüklenir. Şekil 13.2'deki A, B ve C süreçlerini inceleyin; orada her işlemin farklı bir adreste belleğe nasıl yüklendiğini görebilirsiniz. Ve dolayısıyla sorun:

İŞİN PÜF NOKTASI: BELLEK NASIL SANALLAŞTIRILIR

İşletim sistemi, birden fazla çalışan işlem (tümü bellek paylaşımı) için özel, potansiyel olarak büyük bir adres uzayının bu soyutlamasını tek bir fiziksel belleğin üzerine nasıl oluşturabilir?

İşletim sistemi bunu yaptığında, işletim sisteminin **belleği sanallaştırdığını (virtualizing memory)** söylüyoruz, çünkü çalışan program belirli bir adreste (0 diyelim) belleğe yüklendiğini ve potansiyel olarak çok büyük bir adres uzayına (örneğin 32 bit veya 64 bit) sahip olduğunu düşünüyor; gerçek ise oldukça farklı.

Örneğin, Şekil 13.2'deki A işlemi 0 adresinde (**sanal** bir **adres(virtual address)** olarak adlandıracağımız) bir yükleme gerçekleştirmeye çalıştığında, bir şekilde işletim sistemi, bazı donanım destekleriyle birlikte, yüklemenin aslında fiziksel adres 0'a değil, fiziksel adres 320KB'ye (A'nın belleğe yüklendiği) gittiğinden emin olmak zorunda kalacaktır. Bu, dünyadaki her modern bilgisayar sisteminin temelini oluşturan belleği sanallaştırmanın anahtarıdır.

13.4 Hedefler

Böylece, bu not dizisinde işletim sisteminin işine geliyoruz: belleği sanallaştırmak. İşletim sistemi sadece belleği sanallaştırmakla kalmayacak; bunu tarzla yapacak. İşletim sisteminin bunu yaptığından emin olmak için bize rehberlik edecek bazı hedeflere ihtiyacımız var. Bu hedefleri daha önce gördük (Giriş'i düşünün) ve onları tekrar göreceğiz, ancak kesinlikle tekrarlamaya değer.

Bir sanal bellek (VM) sisteminin ana hedeflerinden biri **şeffaflıktır (transparency)**[2]. İşletim sistemi, sanal belleği çalışan program tarafından görülemeyecek 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) birçok farklı iş arasında belleği çoklamak için tüm işi yapar ve bu nedenle yanılsamayı gerçekleştirir.

Sanal belleğin bir diğer amacı **verimliliktir(efficiency)**. İşletim sistemi, hem zaman (yani programları çok daha yavaş çalıştırmamak) hem de alan (yani sanallaştırmayı desteklemek için gereken yapılar için çok fazla bellek kullanmamak) açısından sanallaştırmayı mümkün olduğunca **verimli(efficient)** hale getirmeye çalışmalıdır. Zaman açısından verimli sanallaştırmayı uygularken, İşletim Sisteminin, TLB'ler gibi donanım özellikleri (bu konuyu daha sonra öğreneceğiz) dahil olmak üzere donanım desteğine dayanması gerekecektir.

Son olarak, üçüncü bir sanal makine hedefi **korumadır(protection).** İşletim sistemi, işlemleri birbirinden ve işletim sisteminin kendisini işlemlerden **koruduğundan(protect)** emin olmalıdır.

[2] Bu şeffaflık kullanımı bazen kafa karıştırıcıdır; Bazı öğrenciler "şeffaf olmanın" her şeyi açıkta tutmak anlamına geldiğini yani yönetimin nasıl olması gerektiği anlamına geldiğini düşünüyor. Burada tam tersi bir anlama gelir: İşletim sistemi tarafından sağlanan yanılsamanın uygulamalar tarafından görülmemesi gerektiği. Bu nedenle, yaygın kullanımda şeffaf bir sistem, Bilgi Edinme Özgürlüğü Yasası'nın öngördüğü şekilde taleplere yanıt veren değil, fark edilmesi zor olan sistemdir.

İPUCU: İZOLASYON PRENSİBİ
İzolasyon, güvenilir sistemler oluşturmak için anahtar bir ilkedir. İki varlık birbirinden düzgün bir şekilde izole edilirse, bu birinin diğerini etkilemeden başarısız alabileceğiniz anlamına gelir. İşletim sistemleri, süreçleri birbirinden izole etmeye çalışır ve bu şekilde birinin diğerine zarar vermesini önler. İşletim sistemi, bellek yalıtımını kullanarak, çalışan programların temel alınan 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 yekpare çekirdek tasarımlarından daha fazla güvenilirlik sağlayabilir.

Bir işlem bir yükleme, depo veya talimat getirme işlemi gerçekleştirdiğinde, başka bir işlemin veya işletim sisteminin kendisinin (yani, adres uzayı dışındaki herhangi bir şeyin) bellek içeriğine hiçbir şekilde erişememesi veya bunları etkileyememesi gerekir. Böylece koruma, süreçler arasında yalıtım özelliğini sunmamızı sağlar; Her süreç kendi **izole(isolation)** kozasında çalışmalı, diğer hatalı ve hatta kötü niyetli süreçlerin tahribatından korunmalıdır.

Sonraki bölümlerde, araştırmamızı donanım ve işletim sistemi desteği de dahil olmak üzere belleği sanallaştırmak için gereken temel **mekanizmaları(mechanisms)** odaklanacağız. Ayrıca, boş alanın nasıl yönetileceği ve alanınız azaldığında hangi sayfaların bellekten atılacağı da dahil olmak üzere, işletim sistemlerinde karşılaşacağınız daha alakalı **politikalardan(policies)** bazılarını da araştıracağız. Bunu yaparken, modern bir sanal bellek sisteminin gerçekten nasıl çalıştığına dair anlayışınızı geliştireceğiz [3].

13.5 Özet

Büyük bir işletim sistemi alt sisteminin tanıtıldığını gördük: sanal bellek. Sanal bellek sistemi, tüm talimatlarını ve verilerini burada tutan programlara geniş, seyrek, özel bir adres alanı yanılsaması sağlamaktan sorumludur. İşletim sistemi, bazı ciddi donanım yardımlarıyla, bu sanal bellek referanslarının her birini alacak ve istenen bilgileri almak için fiziksel belleğe sunulabilecek fiziksel adreslere dönüştürecektir. İşletim sistemi bunu birçok işlem için aynı anda yapacak, programları birbirinden korumanın yanı sıra işletim sistemini de koruyacaktır. Tüm yaklaşım, çalışmak için bazı kritik politikaların yanı sıra çok sayıda mekanizma (çok sayıda düşük seviyeli makine) gerektirir; önce kritik mekanizmaları açıklayarak aşağıdan yukarıya başlayacağız. Ve böylece devam ediyoruz!

[3] Ya da sizi süreci bırakmaya ikna edeceğiz. Ama bekle; sanal bellek üzerinden yaparsanız, muhtemelen sonuna kadar başaracaksınız!

BİR BAKIMDAN: GÖRDÜĞÜNÜZ HER ADRES SANALDIR

Hiç pointer (işaretçi) yazdıran bir C programı yazdınız mı? Gördüğünüz değer (genellikle onaltılık olarak yazdırılan bazı büyük sayılar), bir **sanal adrestir (virtual address).** Programınızın kodunun nerede bulunduğunu hiç merak ettiniz mi? Bunu da yazdırabilirsiniz ve evet, yazdırabiliyorsanız, bu aynı zamanda sanal bir adrestir. Aslında, kullanıcı düzeyinde bir programın programcısı olarak görebileceğiniz herhangi bir adres sanal bir adrestir. Bu talimatların ve veri değerlerinin makinenin fiziksel belleğinin neresinde olduğunu bilen, belleği sanallaştırmanın zorlu teknikleri aracılığıyla yalnızca işletim sistemidir. Bu yüzden asla unutmayın: Bir programda bir adres yazdırırsanız, bu sanal bir adrestir, herşeyin bellekte nasıl düzenlendiğine dair bir yanılsamadır; sadece işletim sistemi (ve donanım) doğru gerçeği bilir.

Burada `main()` yordamının (kodun bulunduğu yer) konumlarını, `malloc()`'dan döndürülen heap(yığın) ayrılmış değerin değerini ve stackteki(yığın) bir tamsayının konumunu yazdıran küçük bir program (`va.c`) verilmiştir:

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

64 bit Mac'te çalıştırdığımızda, aşağıdaki çıktıyı alırız:

```
kodun konumu : 0x1095afe50
heap'in konumu : 0x1096008c0
stack'in konumu: 0x7fff691aea64
```

Buradan, kodun önce adres uzayında, sonra heapte(yığın) geldiğini ve stackin(yığın) bu büyük sanal uzayın diğer ucunda olduğunu görebilirsiniz. Bu adreslerin tümü 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 (Kod)

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

Sorular

1. Kontrol etmeniz gereken ilk Linux aracı, `free` çok basit bir araçtır. İlk olarak, `man free` yazın ve tüm kılavuz sayfasını okuyun; kısa, endişelenmeyin!

2. Şimdi, belki de yararlı olabilecek bazı bağımsız değişkenleri kullanarak `free` çalıştırın (örneğin, bellek toplamlarını megabayt cinsinden görüntülemek için `-m`). Sisteminizde ne kadar bellek var? Ne kadarı kullanıma açık? Bu sayılar sezgilerinizle eşleşiyor mu?

```
               total        used        free      shared  buff/cache   available
Mem:            4078        1058        2119          45         900        2737
Swap:           3376           0        3376
```

İlk soruda istenilen free aracının kullanımını öğrendikten sonra bu soruda `free --mega` komutuyla toplam bellek miktarı, kullanılan bellek miktarı ve kullanılabilecek bellek miktarlarını yukarıda yazdırdık.

```
               total        used        free
Mem:            4078        1058        2119
```

Toplam bellek miktarı = 4078mb
Kullanılan bellek miktarı = 1058mb
Kullanıma açık bellek miktarı = 2119mb
Ubuntu programının kurulumunda ram kullanımına 4gb vermiştim bu yüzden bulduğum veriler kendi verilerimle eşleşmektedir.

3. Ardından, `memory-user.c` adı verilen belirli miktarda bellek kullanan küçük bir program oluşturun. Bu program bir komut satırı argümanı almalıdır: kullanacağı megabayt bellek sayısı. Çalıştırıldığında, bir dizi ayırmalı ve her girişe dokunarak dizi boyunca sürekli akış sağlamalıdır. Program bunu süresiz olarak veya belki de komut satırında belirtilen belirli bir süre boyunca yapmalıdır.

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

int main(int argc, char *argv[]) {
  // Eğer kullanıcı bellek miktarını ve programın kaç saniye çalışacağını
  // vermediyse programın kullanımı hakkında bilgi verip sonlandırıyoruz.
  if (argc != 3) {
    printf("Usage: memory-user [memory_in_mb] [duration_in_seconds]\n");
    return 1;
  }

  // Bellek miktarını argümandan alıyoruz.
  int memory_in_mbytes = atoi(argv[1]) * 1024 * 1024;

  // Önceden bellek ayırıyoruz.
  int *array = malloc(memory_in_mbytes);

  // Programın kaç saniye çalışacağını argümandan alıyoruz.
  int duration = atoi(argv[2]);

  // Önceden belirlenen süre boyunca döngü kurarak belleği kullanıyoruz.
  for (size_t i = 0; i < duration; i++) {
    for (size_t j = 0; j < memory_in_mbytes / sizeof(int); j++) {
      array[j]++;
    }
    sleep(1);
  }

  // Belleği serbest bırakıyoruz.
  free(array);

  return 0;
}
```

Yukarıdaki program `memory-user.c` adındaki programdır. Ve bu program belli bir süre boyunca yapılmıştır.

```c
int memory_in_mbytes = atoi(argv[1]) * 1024 * 1024;
```

Yukarıdaki kod parçasında argümandan programın kullanacağı bellek miktarını alır.

```c
if (argc != 3) {
  printf("Usage: memory-user [memory_in_mb] [duration_in_seconds]\n");
  return 1;
}
```

Yukarıdaki kod parçasında kullanıcı bellek miktarı ve programın kaç saniye çalışacağını vermediyse programın kullanımı hakkında bilgi verip program sonlanır.

```c
int *array = malloc(memory_in_mbytes);
```

Yukarıdaki kod parçası bir dizi oluşturmak için bir bellek bölgesi ayrımaya yarar.

```
int duration = atoi(argv[2]);
```

Yukarıdaki kod parçası argümandan programın kaç saniye çalışacağı bilgisini alır.

```
for (size_t i = 0; i < duration; i++) {
  for (size_t j = 0; j < memory_in_mbytes / sizeof(int); j++) {
    array[j]++;
  }
  sleep(1);
}
```

Yukarıdaki parçasında, belirtilen bellekteki dizi elemanlarını belirtilen süre boyunca bir arttırarak işlem yapar.
"duration" parametresi belirtilen süre boyunca bir döngü çalıştırır.
"sleep (1)" ifadesi ise her bir döngü adımından sonra 1 saniye beklemeyi sağlar.

```
free(array);

return 0;
```

Son olarak yukarıdaki kod parçası, bellekte ayrılmış bir dizi bölgesine boşaltarak programın çalışmasını sonlandırır.

4. Şimdi, `memory-user` programınızı çalıştırırken, aynı zamanda (farklı bir terminal penceresinde, ancak aynı makinede) `free` aracını çalıştırın. Programınız çalışırken bellek kullanım toplamları nasıl değişir? `memory-user` programını sonlandırdığınızda ne olur? Rakamlar beklentilerinizi karşılıyor mu? Farklı miktarlarda bellek kullanımı için bunu deneyin. Gerçekten büyük miktarda bellek kullandığınızda ne olur?

```
              total        used        free      shared  buff/cache   available
Mem:           4078        1058        2119          45         900        2737
Swap:          3376           0        3376
```

`memory-user` programını çalıştırmadan önceki bellek değerleri yukarıdadır.

```
se@se-virtual-machine:~/Desktop/memoryuser$ ./memoryuser 1024 5
```

Yukarıda gördüğümüz komutda ./memoryuser 1024 5 komutu ile memoryuser dosyasındaki bizim oluşturduğumuz `memory-user.c` adındaki dosya çalışır ve 1024mb bellek ayırıp 5 saniye boyunca çalışır.

```
se@se-virtual-machine:~$ free --mega
              total        used        free      shared  buff/cache   available
Mem:           4078        2172         983          58         923        1611
Swap:          3376           0        3376
```

Yukarıda ise `memory-user` programı çalışırken alınan bellek değerleri verilmiştir. Burada programın kullandığı alanı eklersek 1058mb önceki kullanılan bellek miktarıydı 1024mb program kullandı toplam 2082mb'lık bir alan kullanılmış oldu. Boş alanda ise

2119mb alan vardı 1024mb kullanıldığı için çıkarıldı 1095mb kullanılabilir alan kalmış oldu. Bu sayılardaki sapan 100mb'a yakın alanlar ise arka planda yapılan işlemlerden dolayı oluşan kayıplardır. Yani rakamlar bizim beklentilerimizi karşılar.

```
              total        used        free      shared  buff/cache   available
Mem:           4078        1097        2057          58         923        2685
Swap:          3376           0        3376
```

Program sonlandırıldıktan sonra alınan değerler yukarıdadır. Değerlere bakacak olursak program sonlandıktan kullanılan alanın azaldığı ve kullanılabilir alanın arttığını yani eski değerlerine yakın değerlere döndüğünü görürüz.

Son olarak çok büyük miktarlarda yani kullanılabilir bellek miktarına yakın değerlerde bellek kullanırsak programın çalışması sırasında sistemde kalan bellek miktarı yetersiz kalabilir. Bu durumda, sistem bellek yetersizliğini gidermek için bellekten yapılan istekleri reddedebilir veya bellek bölgelerini yeniden dağıtabilir. Bu işlemler sırasında programın çalışmasının etkilenmesi veya programın kendisinin durdurulması gibi olumsuz sonuçlar ortaya çıkabilir.

5. `pmap` olarak bilinen bir araç daha deneyelim. Zaman ayırın ve `pmap` kılavuz sayfasını ayrıntılı olarak okuyun.
6. `pmap` 'i kullanmak için, ilgilendiğiniz sürecin **işlem kimliğini (process ID)** bilmeniz gerekir. Bu nedenle, önce tüm işlemlerin bir listesini görmek için `ps auxw` 'u çalıştırın; ardından, tarayıcı gibi ilginç bir tane seçin. Bu durumda `memory-user` programınızı da kullanabilirsiniz (aslında, bu programa `getpid()` çağrısını yaptırabilir ve size kolaylık olması için PID'sini yazdırabilirsiniz).

```
printf("Process ID: %d\n", getpid());
```

`memory-user` programımıza yukarıdaki kod parçasını ekleyip tekrar çalıştırdığımızda bize `memory-user` programının işlem kimliğini (process ID) yazdırır.

```
se@se-virtual-machine:~/Desktop/memoryuser$ ./memoryuser 1024 5
Process ID: 4197
```

Yukarıda `memory-user` programını çalıştırdığımızda işlem kimliğinin 4197 olduğunu görüyoruz.

```
se@se-virtual-machine:~$ ps auxw
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.2 101156 11676 ?        Ss   20:13   0:01 /sbin/init auto noprompt splash
root           2  0.0  0.0      0     0 ?        S    20:13   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   20:13   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   20:13   0:00 [rcu_par_gp]
root           5  0.0  0.0      0     0 ?        I<   20:13   0:00 [netns]
root           7  0.0  0.0      0     0 ?        I<   20:13   0:00 [kworker/0:0H-events_highpri]
root           9  0.0  0.0      0     0 ?        I<   20:13   0:00 [kworker/0:1H-events_highpri]
root          10  0.0  0.0      0     0 ?        I<   20:13   0:00 [mm_percpu_wq]
```

Yukarıda `ps auxw` 'u çalıştırdığımızda çıkan değerlerin bir kısmını belirtmek istedim.

7. Şimdi `pmap'i` bu işlemlerden bazılarında çeşitli işaretler kullanarak çalıştırın (örneğin `-X`) işlemle ilgili birçok ayrıntıyı ortaya çıkarmak için. Ne görüyorsun? Basit kod/stack(yığın)/heap(yığın) anlayışımızın aksine, modern bir adres uzayını kaç farklı varlık oluşturur?

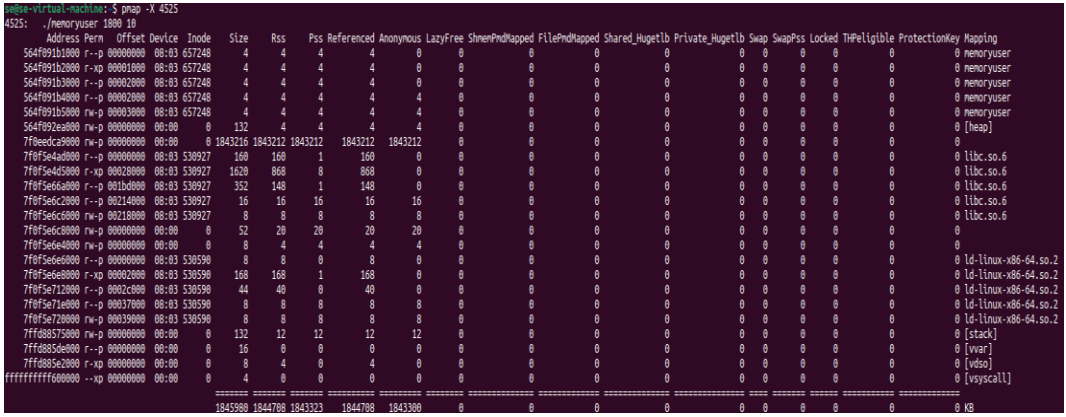
```
se@se-virtual-machine:~/Desktop/memoryuser$ ./memoryuser 1024 10
Process ID: 4479
```

```
se@se-virtual-machine:~$ pmap -X 4479
4479:   ./memoryuser 1024 10
         Address Perm   Offset Device  Inode   Size    Rss    Pss Referenced Anonymous LazyFree ShmemPmdMapped FilePmdMapped Shared_Hugetlb Private_Hugetlb Swap SwapPss Locked THPeligible ProtectionKey Mapping
    55870c19f000 r--p 00000000 08:03 657248      4      4      4          4         0        0              0             0              0              0    0       0      0           0             0 memoryuser
    55870c1a0000 r-xp 00001000 08:03 657248      4      4      4          4         0        0              0             0              0              0    0       0      0           0             0 memoryuser
    55870c1a1000 r--p 00002000 08:03 657248      4      4      4          4         0        0              0             0              0              0    0       0      0           0             0 memoryuser
    55870c1a2000 r--p 00002000 08:03 657248      4      4      4          4         4        0              0             0              0              0    0       0      0           0             0 memoryuser
    55870c1a3000 rw-p 00003000 08:03 657248      4      4      4          4         4        0              0             0              0              0    0       0      0           0             0 memoryuser
    55870c3aa000 rw-p 00000000 00:00      0    132      4      4          4         4        0              0             0              0              0    0       0      0           0             0 [heap]
    7fe4629c8000 rw-p 00000000 00:00      0 1048592 1048588 1048588    1048588   1048588        0              0             0              0              0    0       0      0           0             0
    7fe4a29cc000 r--p 00000000 08:03 530927    160    160      1        160         0        0              0             0              0              0    0       0      0           0             0 libc.so.6
    7fe4a29f4000 r-xp 00028000 08:03 530927   1620    936      9        936         0        0              0             0              0              0    0       0      0           0             0 libc.so.6
    7fe4a2b89000 r--p 001bd000 08:03 530927    352    152      1        152         0        0              0             0              0              0    0       0      0           0             0 libc.so.6
    7fe4a2be1000 r--p 00214000 08:03 530927     16     16     16         16        16        0              0             0              0              0    0       0      0           0             0 libc.so.6
    7fe4a2be5000 rw-p 00218000 08:03 530927      8      8      8          8         8        0              0             0              0              0    0       0      0           0             0 libc.so.6
    7fe4a2be7000 rw-p 00000000 00:00      0     52     20     20         20        20        0              0             0              0              0    0       0      0           0             0
    7fe4a2c03000 rw-p 00000000 00:00      0      8      4      4          4         4        0              0             0              0              0    0       0      0           0             0
    7fe4a2c05000 r--p 00000000 08:03 530590      8      8      0          8         0        0              0             0              0              0    0       0      0           0             0 ld-linux-x86-64.so.2
    7fe4a2c07000 r-xp 00002000 08:03 530590    168    168      1        168         0        0              0             0              0              0    0       0      0           0             0 ld-linux-x86-64.so.2
    7fe4a2c31000 r--p 0002c000 08:03 530590     44     40      0         40         0        0              0             0              0              0    0       0      0           0             0 ld-linux-x86-64.so.2
    7fe4a2c3d000 r--p 00037000 08:03 530590      8      8      8          8         8        0              0             0              0              0    0       0      0           0             0 ld-linux-x86-64.so.2
    7fe4a2c3f000 rw-p 00039000 08:03 530590      8      8      8          8         8        0              0             0              0              0    0       0      0           0             0 ld-linux-x86-64.so.2
    7fff02757000 rw-p 00000000 00:00      0    132     12     12         12        12        0              0             0              0              0    0       0      0           0             0 [stack]
    7fff02789000 r--p 00000000 00:00      0     16      0      0          0         0        0              0             0              0              0    0       0      0           0             0 [vvar]
    7fff0278d000 r-xp 00000000 00:00      0      8      4      0          4         0        0              0             0              0              0    0       0      0           0             0 [vdso]
ffffffffff600000 --xp 00000000 00:00      0      4      0      0          0         0        0              0             0              0              0    0       0      0           0             0 [vsyscall]
                                                ====== ====== ======== ======== ======= ============ =========== ============ ============== ==== ====== ===== ========== ============
                                              1051356 1050156 1048700   1050156   1048676        0              0             0              0              0    0       0      0           0             0 KB
```

Yukarıdaki görüntülerde `memory-user` programını çalıştırıp işlem kimliği 4479 olarak bulunduktan sonra pmap `-X` 4479 komutunu çalıştırdığımızda oluşan ekran bulunuyor.
Bu veriler, çalışan bir uygulamanın adres uzayındaki bellek bölümlerini göstermektedir. Yani buradaki heap, stack ve kod satırları da dahil diğer tüm satırlar adres uzayındaki bellek bölümlerini gösterir.

Bu verilerini her bir satırı, bir bellek bölümünün adres bilgilerini, bağlantılı dosyayı (eğer varsa), boyut bilgilerinin ve diğer özelliklerini gösterir.

8. Son olarak, `memory-user` programınızda, farklı miktarlarda kullanılan bellekle `pmap`'i çalıştıralım. Burada ne görüyorsunuz? `pmap` 'ten elde edilen çıktı beklentilerinizi karşılıyor mu?



```
os@os-virtual-machine: $ pmap -X 4525
4525:   ./memoryuser 1800 10
         Address Perm   Offset Device  Inode     Size      Rss      Pss Referenced Anonymous LazyFree ShmemPmdMapped FilePmdMapped Shared_Hugetlb Private_Hugetlb Swap SwapPss Locked THPeligible ProtectionKey Mapping
    564f891b1000 r--p 00000000 08:03 657248        4        4        4          4         0        0              0             0              0               0    0       0      0             0             0 memoryuser
    564f891b2000 r-xp 00001000 08:03 657248        4        4        4          4         0        0              0             0              0               0    0       0      0             0             0 memoryuser
    564f891b3000 r--p 00002000 08:03 657248        4        4        4          4         0        0              0             0              0               0    0       0      0             0             0 memoryuser
    564f891b4000 r--p 00002000 08:03 657248        4        4        4          4         4        0              0             0              0               0    0       0      0             0             0 memoryuser
    564f891b5000 rw-p 00003000 08:03 657248        4        4        4          4         4        0              0             0              0               0    0       0      0             0             0 memoryuser
    564f892ea000 rw-p 00000000 00:00      0      132        4        4          4         4        0              0             0              0               0    0       0      0             0             0 [heap]
    7f0eedca9000 rw-p 00000000 00:00      0  1843216  1843212  1843212    1843212   1843212        0              0             0              0               0    0       0      0             0             0
    7f0f5e4ad000 r--p 00000000 08:03 530927      160      160        1        160         0        0              0             0              0               0    0       0      0             0             0 libc.so.6
    7f0f5e4d5000 r-xp 00028000 08:03 530927     1620      868        8        868         0        0              0             0              0               0    0       0      0             0             0 libc.so.6
    7f0f5e66a000 r--p 001bd000 08:03 530927      352      148        1        148         0        0              0             0              0               0    0       0      0             0             0 libc.so.6
    7f0f5e6c2000 r--p 00214000 08:03 530927       16       16       16         16        16        0              0             0              0               0    0       0      0             0             0 libc.so.6
    7f0f5e6c6000 rw-p 00218000 08:03 530927        8        8        8          8         8        0              0             0              0               0    0       0      0             0             0 libc.so.6
    7f0f5e6c8000 rw-p 00000000 00:00      0       52       20       20         20        20        0              0             0              0               0    0       0      0             0             0
    7f0f5e6e4000 rw-p 00000000 00:00      0        8        4        4          4         4        0              0             0              0               0    0       0      0             0             0
    7f0f5e6e6000 r--p 00000000 08:03 530590        8        8        0          8         0        0              0             0              0               0    0       0      0             0             0 ld-linux-x86-64.so.2
    7f0f5e6e8000 r-xp 00002000 08:03 530590      168      168        1        168         0        0              0             0              0               0    0       0      0             0             0 ld-linux-x86-64.so.2
    7f0f5e712000 r--p 0002c000 08:03 530590       44       40        0         40         0        0              0             0              0               0    0       0      0             0             0 ld-linux-x86-64.so.2
    7f0f5e71e000 r--p 00037000 08:03 530590        8        8        8          8         8        0              0             0              0               0    0       0      0             0             0 ld-linux-x86-64.so.2
    7f0f5e720000 rw-p 00039000 08:03 530590        8        8        8          8         8        0              0             0              0               0    0       0      0             0             0 ld-linux-x86-64.so.2
    7ffd88575000 rw-p 00000000 00:00      0      132       12       12         12        12        0              0             0              0               0    0       0      0             0             0 [stack]
    7ffd885de000 r--p 00000000 00:00      0       16        0        0          0         0        0              0             0              0               0    0       0      0             0             0 [vvar]
    7ffd885e2000 r-xp 00000000 00:00      0        8        4        0          4         0        0              0             0              0               0    0       0      0             0             0 [vdso]
ffffffffff600000 --xp 00000000 00:00      0        4        0        0          0         0        0              0             0              0               0    0       0      0             0             0 [vsyscall]
                                                ====== ======= ======= ========== ========= ======== ============== ============= ============== =============== ==== ======= ====== ========= ===========
                                              1845980 1844708 1843323    1844708   1843300        0              0             0              0               0    0       0      0             0             0 KB
```

Bir önceki soruda 1024mb bellek kullanan `memory-user` programının ekran görüntüsü vardı. Bu soruda verilen ekran görüntüsünde ise 1800mb bellek kullanan `memory-user` programının ekran görüntüsü bulunmaktadır.

İkisininde modern adres uzayını aynı sayıda varlık oluşturur. Fakat boyut da dahil olmak üzere tüm özelliklerde 1800mb bellek kullanan `memory-user` programı değerleri daha büyüktür. Bu çıktılar bizim öngördüğümüz çıktılarla uyuşmaktadır.