13

Soyutlama: Adres Uzayları

İlk günlerde bilgisayar sistemleri oluşturmak kolaydı. Neden diye soracaksınız ? Çünkü kullanıcılar fazla bir şey beklemiyordu. Tüm bu baş ağrılarına gerçekten "kullanım kolaylığı", "yüksek performans", "güvenilirlik" vb. beklentileri olan lanetlenmiş kullanıcılar neden olmuştur. Bir dahaki sefere bu bilgisayar kullanıcılarından biriyle karşılaştığınızda, neden oldukları tüm sorunlar için onlara teşekkür edin.

13.1 Erken Dönemlerdeki Sistemler

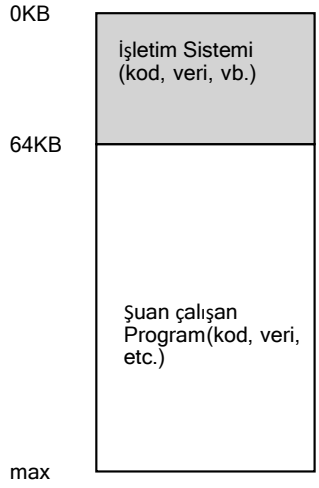
Bellek açısından bakıldığında, ilk makineler kullanıcılara çok fazla soyutlama sağlamıyordu. Temel olarak, makinenin fiziksel belleği Şekil 13.1'de (Sayfa 2) gördüğünüz gibi görünüyordu.

İşletim sistemi(OS), bellekte oturan (bu örnekte fiziksel adresi 0'dan başlayan) bir dizi rutindi (aslında bir kitaplık) ve halihazırda fiziksel bellekte(bu örnekte 64k fiziksel adresinden başlıyor) oturan ve belleğin kalanını kullanan tek çalışan **program(bir işlem(a process))** olacaktı.Burada biraz yanılsama vardı,ve kullanıcı **işletim sisteminden(OS**) pek birşey beklemiyordu. İşletim sistemi geliştiricileri için o günlerde 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 başladılar. Böylece, birden çok işlemin belirli bir zamanda çalışmaya hazır olduğu ve örneğin bir **G/Ç(I/O)** gerçekleştirmeye karar verildiğinde **işletim sisteminin(OS)** bunlar arasında geçiş yaptığı **çoklu programlama(multiprogramming)** çağı [DV66] doğdu.Bunu yapmak, CPU'nun **etkin kullanımını(utilization)** artırdı. **Verimlilikteki(efficiency)** bu tür artışlar, her makinenin yüz binlerce hatta milyonlarca dolara mal olduğu (ve siz Mac'inizin pahalı olduğunu düşünüyorsunuz) 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 kendilerinde [CV65], **toplu hesaplamanın(batch computing**) sınırlamalarını fark etti.



Şekil 13.1: **İşletim Sistemleri: Eski zamanlarda**

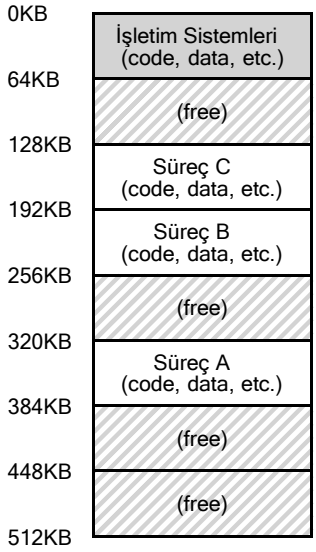
Etkileşim(interactivity) kavramı önemli hale geldi, çünkü birçok kullanıcı aynı anda bir makineyi kullanıyor olabilir ve her biri o sırada yürütmekte oldukları **görevlerden(process)** zamanında bir yanıt bekler (veya bunu umar).

Zaman paylaşımını(time sharing) uygulamanın bir yolu, bir **işlemi(process)** kısa bir süre çalıştırarak ona tüm **belleğe(memory)** tam erişim vermek (Şekil 13.1), ardından durdurmak, tüm durumunu bir tür diske kaydetmek (tüm fiziksel bellek dahil olabilir), başka bir **işlemin(process**) durumunu yüklemek, onu bir süre çalıştırmak ve böylece makinenin bir tür kaba paylaşımını gerçekleştirmektir. [M+63].

Ne yazık ki, bu yaklaşımın büyük bir sorunu var: özellikle **hafıza(memory)** alanı büyüdükçe çok yavaşlar. Kayıt düzeyinde durumun (PC, genel amaçlı kayıtlar, vb.) kaydedilmesi ve geri yüklenmesi nispeten hızlı olsa da, **belleğin(memory)** tüm içeriğini diske kaydetmek acımasızca performanssızdır. Bu nedenle, yapmayı tercih ettiğimiz şey, **işlemler(processes)** arasında geçiş yaparken **işlemleri(process) bellekte(memory)** bırakarak **işletim sisteminin(OS) zaman paylaşımını(time sharing)** verimli bir şekilde uygulamasına izin vermektir (Şekil 13.2, sayfa 3'te gösterildiği gibi).

Diyagramda üç **işlem(process)** (A, B ve C) vardır ve her biri kendileri için oyulmuş 512 KB'lik fiziksel belleğin küçük bir kısmına sahiptir. Tek bir CPU varsayarsak, **işletim sistemi(OS) süreçlerden(process)** birini (örneğin A) çalıştırmayı seçerken, diğerleri (B ve C) hazır kuyruğunda çalışmayı bekler.

Zaman paylaşımı(time-sharing) daha popüler hale geldikçe, muhtemelen **işletim sistemine(OS)** yeni talepler getirildiğini tahmin edebilirsiniz. Özellikle birden çok programın aynı anda bellekte bulunmasına izin vermek, korumayı önemli bir konu haline getirir; bir **işlemin(process)** başka bir işlemin hafızasını okuyabilmesini veya daha kötüsü yazabilmesiniistemezsiniz.



Şekil 13.2: **Üç Süreç: Hafıza paylaşımı**

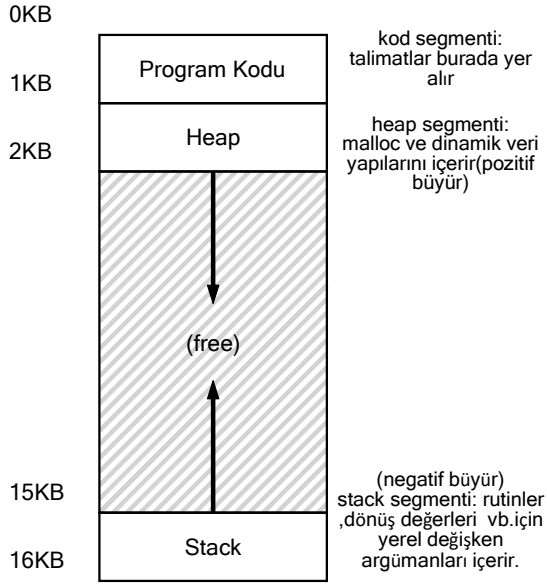
13.3 Adres alanı

Ancak, bu can sıkıcı kullanıcıları aklımızda tutmalıyız ve bunu yapmak için **işletim sisteminin(OS)** kullanımı kolay bir fiziksel bellek soyutlaması oluşturması gerekir. Bu soyutlamaya **adres alanı(adress space)** diyoruz ve çalışan programın sistemdeki **bellekte(memory)** olan görünümüdür. Belleğin bu temel **işletim sistemi(OS)** soyutlamasını anlamak, **belleğin(memory)** nasıl **sanallaştırıldığını(virtualization)** anlamanın anahtarıdır.

Bir işlemin **adres alanı(address space),** çalışan programın tüm bellek durumunu içerir. Örneğin, programın kodu (talimatlar) **bellekte(memory)** bir yerde yaşamak zorundadır ve bu nedenle **adres alanında(address space)** bulunurlar. Program çalışırken, işlev çağrı zincirinde nerede olduğunu takip etmek ve ayrıca yerel değişkenleri tahsis etmek ve rutinlerden parametreleri iletmek ve değerleri döndürmek için bir **yığın(stack)** kullanır. Son olarak, **öbek(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 başlatılan değişkenler), ancak şimdilik bu üç bileşeni varsayalım: kod, **yığın(stack)** ve **öbek(heap).**

Figür 13.3'teki (Sayfa 4) örnekte çok küçük bir adres alanımız(sadece 16KB) [1]var.

[1]Sıklıkla bunun gibi örnekler kullanacağız. Çünkü (a) 32 bit adres alanını temsil etmek zahmetli ve (b) matematik zor. Basit matematiği seviyoruz.



Şekil 13.3: **Adres Space Örneği**

Program kodu, **adres alanının(address space)** en üstünde yer alır (bu örnekte 0'dan başlayarak ve **adres alanının(address space)** ilk 1K'lık kısmına sıkıştırılmıştır). Kod statiktir (ve bu nedenle belleğe yerleştirilmesi kolaydır), bu nedenle onu **adres alanının(address space)** en üstüne yerleştirebilir ve program çalışırken daha fazla alana ihtiyaç duymayacağını bilebiliriz..

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

Elbette **adres alanını(address space)** tarif ettiğimizde, tarif ettiğimiz şey **işletim sisteminin(OS)** çalışan programa sağladığı **soyutlamadır(abstraction**). Program, 0 ile 16KB arasındaki fiziksel adreslerde gerçekten **bellekte(memory**) değildir; bunun yerine bazı rasgele fiziksel adres(ler)e yüklenir. Şekil 13.2'deki A, B ve C süreçlerini inceleyin;

orada her işlemin farklı bir adreste belleğe nasıl yüklendiğini görebilirsiniz. Ve dolayısıyla sorun:

ÖNEMLİ NOKTA: BELLEĞİ NASIL Sanallaştırırız ?
İşletim sistemi, birden çok çalışan işlem (tüm paylaşım belleği) için tek bir fiziksel belleğin üzerinde özel, potansiyel olarak büyük bir adres alanının bu soyutlamasını nasıl oluşturabilir?

İşletim sistemi(OS) bunu yaptığında, **işletim sisteminin(OS) belleği(memory)** sanallaştırdığını(virtualization) söylüyoruz, çünkü çalışan program belleğe belirli bir adreste (0 diyelim) yüklendiğini ve potansiyel olarak çok geniş bir adres alanına sahip (örneğin 32-bit veya 64-bit) olduğunu düşünüyor; gerçeklik oldukça farklıdır.

Örneğin, Şekil 13.2'deki A işlemi, 0 adresinde (biz buna sanal adres diyeceğiz) bir yük(load) gerçekleştirmeye çalıştığında, bir şekilde **işletim sistemi(OS),** bazı donanım desteğiyle birlikte, yükün aslında fiziksel adres 0'a değil, fiziksel adres 320KB'ye (A'nın belleğe yüklendiği yer) gitmediğinden emin olmak zorunda kalacak. Bu, dünyadaki her modern bilgisayar sisteminin temelini oluşturan **belleği(memory) sanallaştırmanın(virtualization)** anahtarıdır.

13.4 Hedefler

Böylece, bu not dizisinde **işletim sisteminin(OS)** işine geliyoruz: belleği **sanallaştırmak(virtualization).** Ancak **işletim sistemi(OS)** yalnızca belleği sanallaştırmaz; bunu bir stil ile yapacak. **İşletim sisteminin(OS)** bunu yaptığından emin olmak için bize rehberlik edecek bazı hedeflere ihtiyacımız var. Bu hedefleri daha önce gördük (Giriş'i düşünün) ve tekrar göreceğiz, ancak kesinlikle tekrar etmeye değer.

Sanal bellek (VM) sisteminin ana hedeflerinden biri şeffaflıktır(**transparency**)[2]. **İşletim sistemi(OS) sanal belleği(virtual memory)**,çalışan program tarafından görülemeyecek şekilde uygulamalıdır. Bu nedenle program, belleğin **sanallaştırıldığının(virtualization)** farkında olmamalıdır; bunun yerine, program kendi özel fiziksel belleğine sahipmiş gibi davranır. Perde arkasında, işletim sistemi (ve donanım) belleği birçok farklı iş arasında çoğullamak için tüm işi yapar ve bu nedenle yanılsamayı gerçekleştirir.

VM'nin bir diğer amacı da verimliliktir. **İşletim sistemi(OS),** hem zaman (yani programları çok daha yavaş çalıştırmamak) hem de alan (yani sanallaştırmayı desteklemek için gereken yapılar için çok fazla bellek kullanmamak) açısından **sanallaştırmayı(virtualization)** olabildiğince verimli hale getirmeye çalışmalıdır. Zaman açısından verimli sanallaştırmayı uygularken, İşletim Sistemi, TLB'ler gibi (ileride öğreneceğimiz) donanım özellikleri de dahil olmak üzere donanım desteğine güvenmek zorunda kalacaktır.

[2]Bu şeffaflık kullanımı bazen kafa karıştırıcıdır; bazı öğrenciler "şeffaf olmanın" her şeyi açıkta tutmak, yani hükümetin nasıl olması gerektiği anlamına geldiğini düşünüyor..
Burada tam tersi bir anlama gelir: İşletim sistemi tarafından sağlanan yanılsamanın uygulamalar tarafından görülmemesi gerekir.
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 PRENSİPLERİ
İzolasyon, güvenilir sistemler oluşturmak için temel bir ilkedir. İki varlık birbirinden düzgün bir şekilde izole edilirse, bu, birinin diğerini etkilemeden başarısız olabileceği anlamına gelir. İşletim sistemleri, süreçleri birbirinden izole etmeye ve bu şekilde birinin diğerine zarar vermesini engellemeye çalışır. İşletim sistemi, bellek yalıtımını kullanarak ayrıca çalışan programların temeldeki 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(mikrokernels**) [BH70, R+89, S+03] bu nedenle tipik yekpare çekirdek tasarımlarından daha fazla güvenilirlik sağlayabilir..

Son olarak, üçüncü bir **sanal makine(virtual machine)** hedefi korumadır. **İşletim sistemi(OS),** işlemleri birbirinden ve işletim sisteminin kendisini işlemlerden koruduğundan emin olmalıdır. Bir işlem bir yükleme, depolama veya talimat getirme işlemi gerçekleştirdiğinde, başka bir işlemin veya işletim sisteminin kendisinin (yani, adres alanı dışındaki herhangi bir şeyin) bellek içeriğine erişememeli veya bunları hiçbir şekilde etkileyememelidir. Böylece koruma, süreçler arasında yalıtım özelliğini sunmamızı sağlar; her süreç, diğer hatalı ve hatta kötü niyetli süreçlerin tahribatından uzak, kendi yalıtılmış kozası içinde çalışmalıdır.

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

13.5 Özet

Büyük bir işletim sistemi alt sisteminin tanıtıldığını gördük: sanal bellek. VM sistemi, tüm talimatlarını ve verilerini burada tutan programlara geniş, seyrek, özel bir adres alanı yanılsaması sağlamaktan sorumludur. İşletim sistemi, bazı ciddi donanım yardımı ile, bu sanal bellek referanslarının her birini alacak ve bunları, istenen bilgileri almak için fiziksel belleğe sunulabilecek fiziksel adreslere dönüştürecektir. İşletim sistemi, programları birbirinden korumanın yanı sıra işletim sistemini de koruyarak bunu birçok işlem için aynı anda yapacaktır. 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 dersi bırakmaya ikna edeceğiz. Ama bekle; sanal makine aracılığıyla başarırsanız, büyük ihtimalle sonuna kadar gidersiniz..

KENAR: GÖRDÜĞÜNÜZ HER ADRES SANALDIR

Hiç işaretçi yazdıran bir C programı yazdınız mı? Gördüğünüz değer (bazı büyük sayılar, genellikle onaltılık olarak yazdırılır), sanal bir adrestir. Programınızın kodunun nerede bulunduğunu hiç merak ettiniz mi? Bunu da yazdırabilirsiniz ve evet, yazdırabilirseniz, bu aynı zamanda bir sanal 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 sanallaştırmaya yönelik kurnaz teknikleri sayesinde yalnızca işletim sistemidir. Bu yüzden asla unutmayın: Bir programda bir adres yazdırırsanız, bu sanal bir adrestir, her şeyin bellekte nasıl düzenlendiğine dair bir yanılsamadır; yalnızca işletim sistemi (ve donanım) gerçek gerçeği bilir.

İşte main() rutininin konumlarını (kodun bulunduğu yer), malloc()'tan döndürülen öbek(heap) tahsisli değerin değerini ve yığındaki(stack) 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'te çalıştırdığımızda aşağıdaki çıktıyı alıyoruz :

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

Buradan, kodun önce adres alanından(adress space), sonra öbekten(heap) geldiğini ve yığının(stack) bu büyük sanal alanın diğer ucunda olduğunu görebilirsiniz. Bu adreslerin tümü sanaldır ve değerleri gerçek fiziksel konumlarından almak için işletim sistemi ve donanım tarafından çevrilecektir.

Referanslar

[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.*

Ev ödevi (Kod)

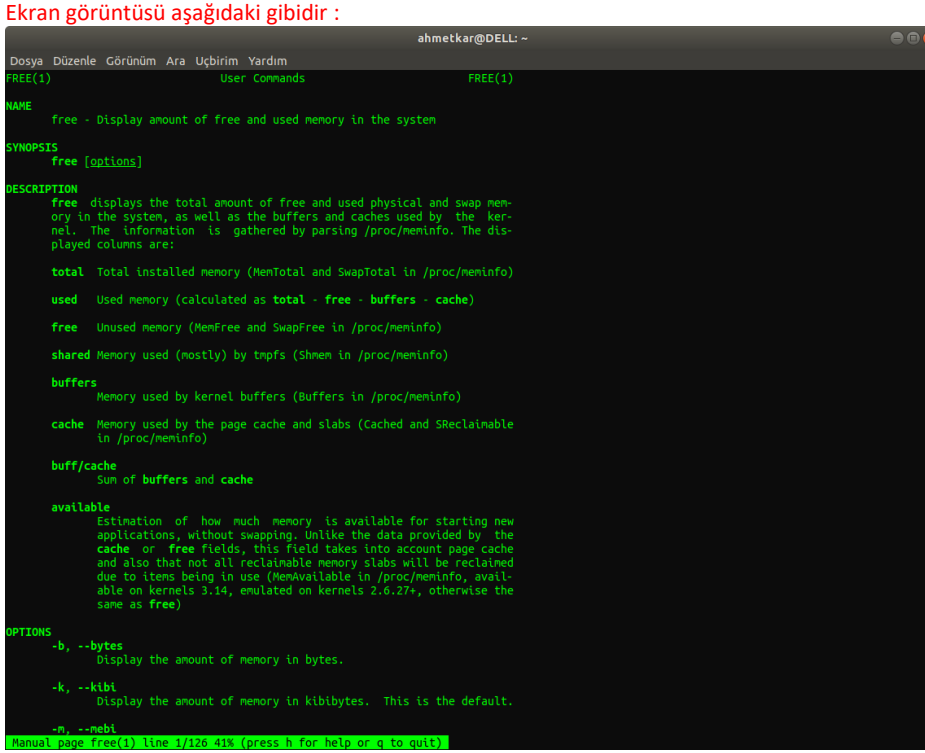
Bu ödevde, Linux tabanlı sistemlerde sanal bellek kullanımını incelemek için birkaç yararlı araç hakkında bilgi edineceğiz. Bu, neyin mümkün olduğuna dair yalnızca kısa bir ipucu olacaktır; gerçekten bir uzman olmak için kendi başınıza daha derine dalmanız gerekecek (her zaman olduğu gibi !)

Sorular

1. Kontrol etmeniz gereken ilk Linux aracı, ücretsiz çok basit bir araçtır. İlk olarak, "man free" yazın ve kılavuz sayfasının tamamını okuyun; kısa oldu, merak etmeyin!

man free free hakkında ve alabileceği parametreler hakkında bilgi verir .
Örneğin free kullanılan ve kullanılmayan fiziksel ve swap bellek miktarını görüntüler.

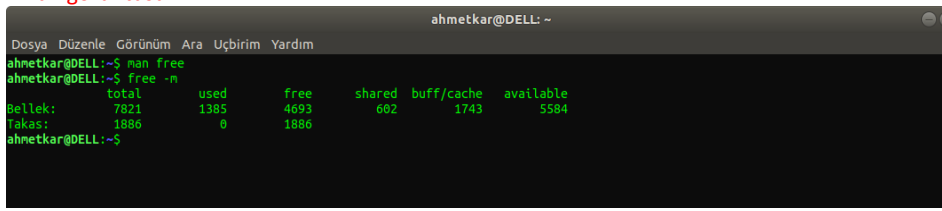
Ekran görüntüsü aşağıdaki gibidir :



2. Şimdi, belki yararlı olabilecek bazı argümanları kullanarak "run free" (örneğin, -m, bellek toplamlarını megabayt cinsinden görüntülemek için). Sisteminizde ne kadar bellek var? Ne kadarı serbest? Bu sayılar sezginize uyuyor mu??

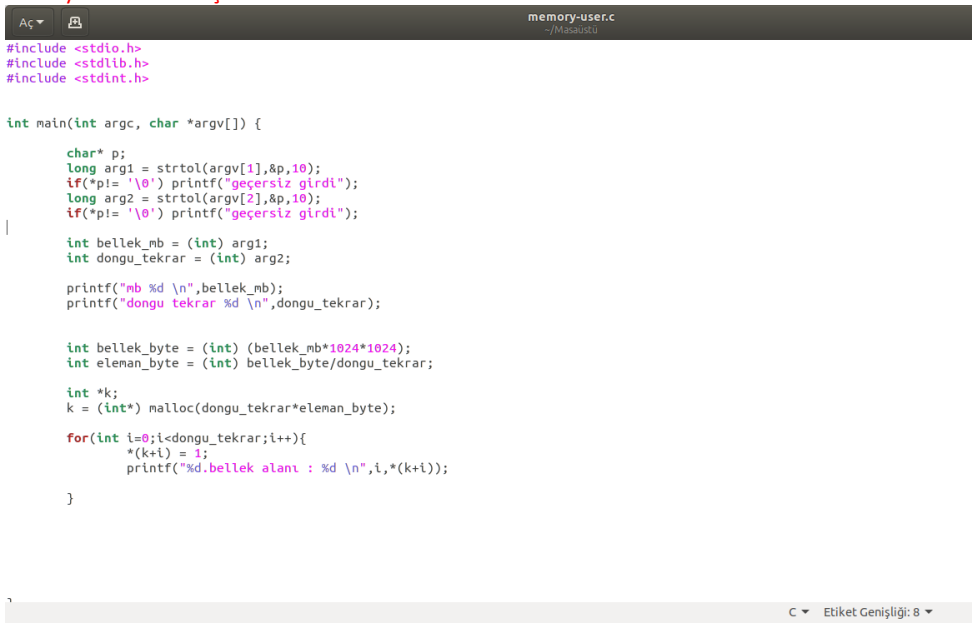
Resimde görüldüğü gibi toplam 7821 mb bellek var şuan 1385 mb kullanılıyor 4693 mb si serbest.Sezgilerime göre kullanılan belleğin daha az ,kullanılmayan belleğin daha fazla olmasını beklerdim.

Ekran görüntüsü :

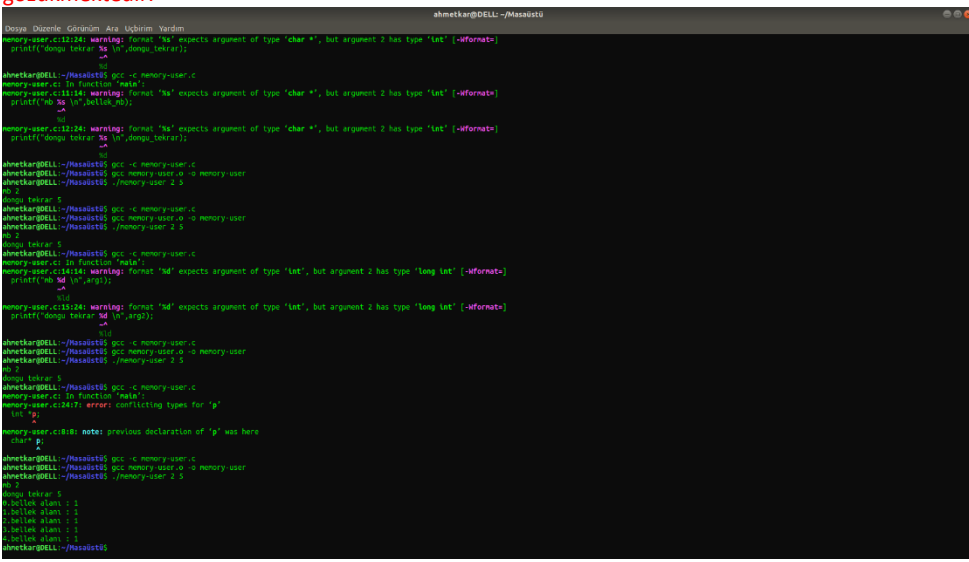


3. Ardından, "memory-user.c" adlı, belirli bir miktarda bellek kullanan küçük bir program oluşturun. Bu program bir komut satırı argümanı almalıdır: kullanacağı megabayt bellek sayısı. Çalıştırıldığında, bir dizi tahsis etmeli ve her girişe dokunarak dizi boyunca sürekli akış yapmalıdır. Program bunu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

İlk ekran görüntüsünde görüldüğü gibi ilk argüman oluşturulacak belleğin toplam kullanacağı alan sayısını,ikinci argüman ise döngü tekrar sayısını ve toplam eleman sayısını almıştır.Böylece ilk olarak char olarak alınan argümanların önce long sonra int tipine dönüşümü sağlanmış sonra bellekte bir dizi için alan tahsisi için malloc fonksiyonu kullanılmıştır.

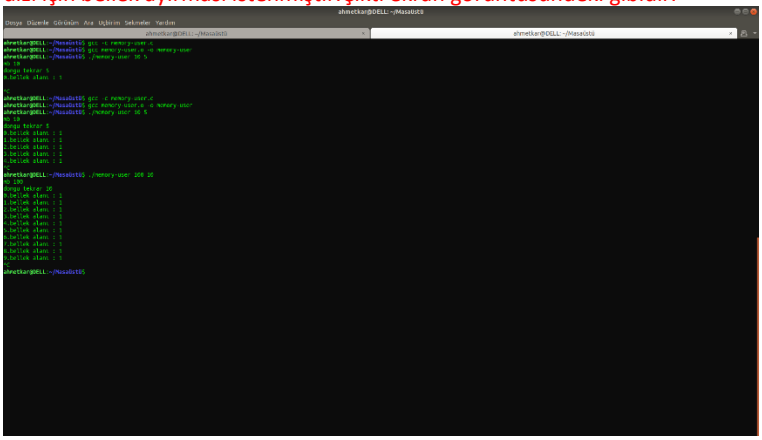


İkinci ekran görüntüsünde kodun çıktısını görüyoruz. 2 mb lık toplam alanda 5 elemana alan tahsisi yapılmış hepsinin değeri teker teker 1 e eşitlenmiştir.Çıktıda bu gözükmektedir.

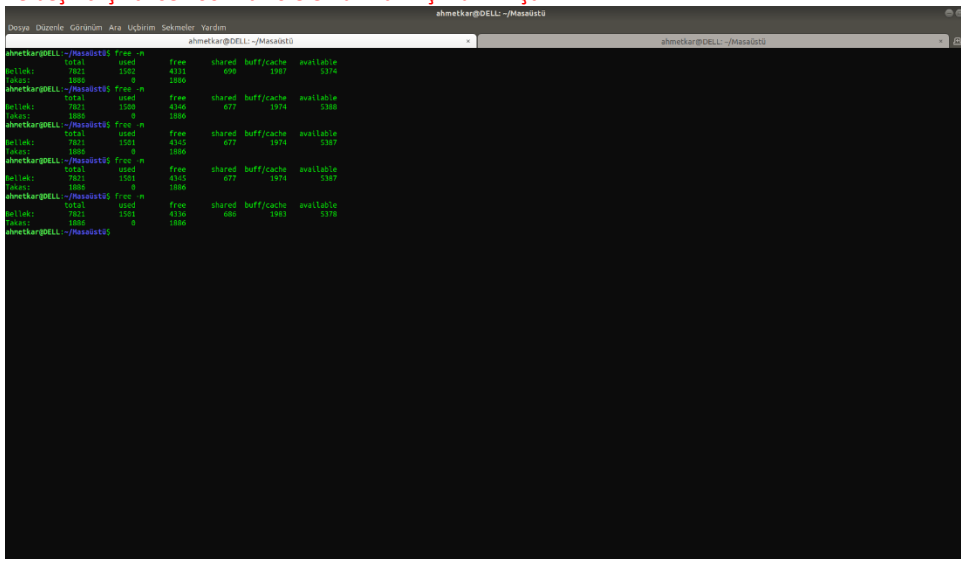


4. Şimdi, "memory-user" programınızı çalıştırırken (farklı bir terminal penceresinde, ancak aynı makinede) "free" aracını da çalıştırın. Programınız çalışırken bellek kullanım toplamları nasıl değişir? "memory-user" programını sonlandırdığınızda ne olur? Rakamlar beklentilerinizle örtüşüyor mu? Farklı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

Programa önce 10 mb lık 5 elemanı dizi için bellek ayırması sonra 100 mb lik 10 elemanlı dizi için bellek ayırması istenmiştir.Çıktı ekran görüntüsündeki gibidir.



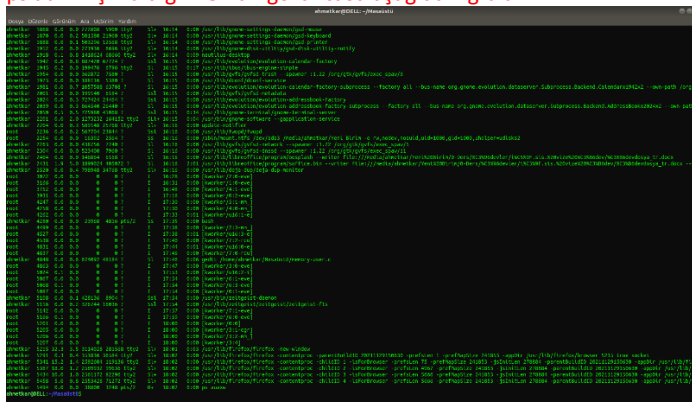
Free komutunun çıktısı ise aşağıdaki ekran görüntüsündeki gibidir. İlk çıktı hiçbir şey yapılmadan öncesiyken sırasıyla ikinci ve üçüncü çıktı 10 mb 5 elemanlı dizi için,dördüncü ve beşinci çıktı ise 100 mb 10 elemanlı dizi için alınmıştır.



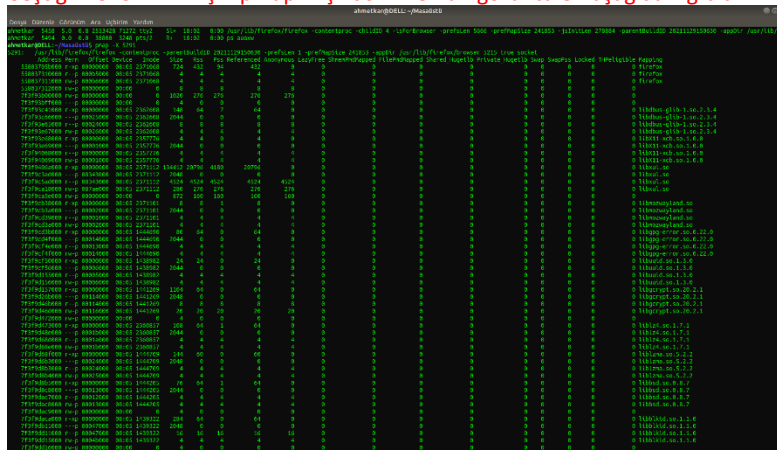
5. "pmap" olarak bilinen bir araç daha deneyelim. Biraz zaman ayırın ve "pmap" kılavuz sayfasını ayrıntılı olarak okuyun.

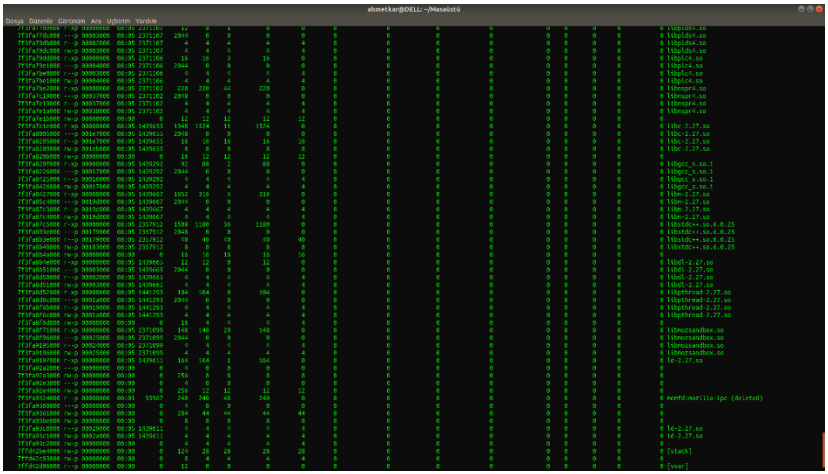
man pmap ile pmap kılavuz sayfası okunmuştur.pmap argüman olarak aldığı process veya processler için detaylı bir memory bilgisi sunar.

6. "pmap" kullanmak için, ilgilendiğiniz işlemin process id'sini bilmeniz gerekir. Bu nedenle, tüm işlemlerin bir listesini görmek için önce "ps auxw" komutunu çalıştırın; ardından tarayıcı gibi ilginç bir tane seçin. Bu durumda "memory-user" programınızı da kullanabilirsiniz (aslında, bu programın "getpid()" aramasını ve size kolaylık olması için PID'sini yazdırmasını bile sağlayabilirsiniz).
ps auxw için aldığım ekran görüntüsü aşağıdaki gibidir.



7. Şimdi, süreçle ilgili birçok ayrıntıyı ortaya çıkarmak için çeşitli bayrakları (-X gibi) kullanarak bu süreçlerin bazılarında "pmap" çalıştırın. Ne görüyorsun? Basit kod/yığın/öbek anlayışımızın aksine, modern bir adres alanını kaç farklı varlık oluşturur?
Seçtiğim 5291 PID i için pmap in çıktısının ekran görüntüleri aşağıdaki gibidir.

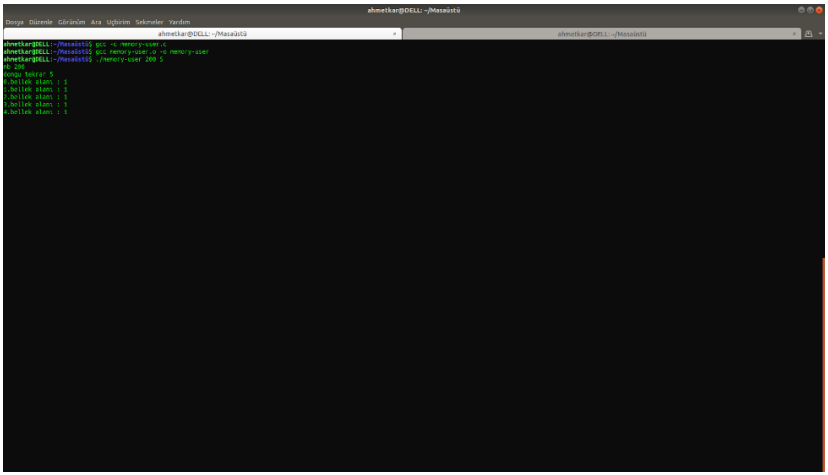




Sadece stack,heap,kod bellek alanı değil birçok varlık için bellek alanı ayrılmış olduğunu görüyoruz.

8. Son olarak "memory-user" programınızda "pmap"i farklı miktarlarda kullanılmış bellekle çalıştıralım. Burada ne görüyorsunuz? "pmap" çıktısı beklentilerinizi karşılıyor mu??

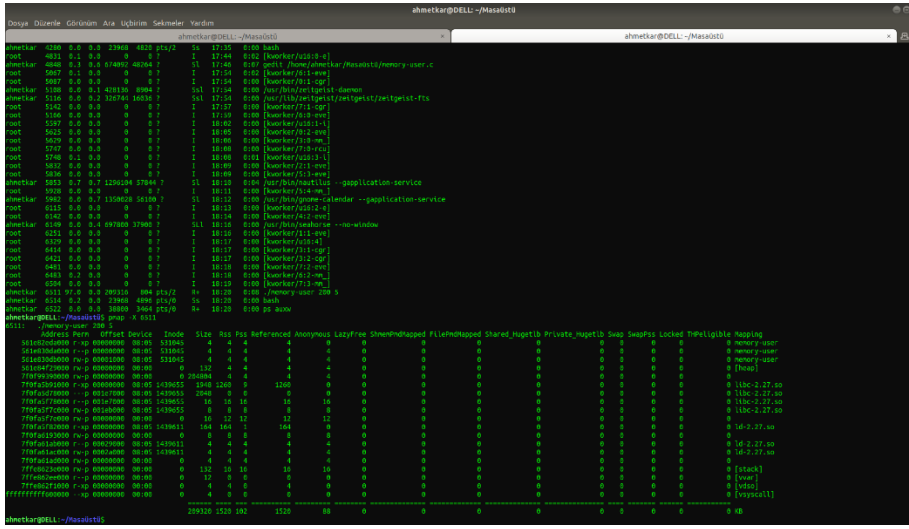
Memory-user programını ilk olarak 200 mb 5 eleman girdileriyle çalıştırdık ve ekran görüntüsü aşağıdaki gibidir.



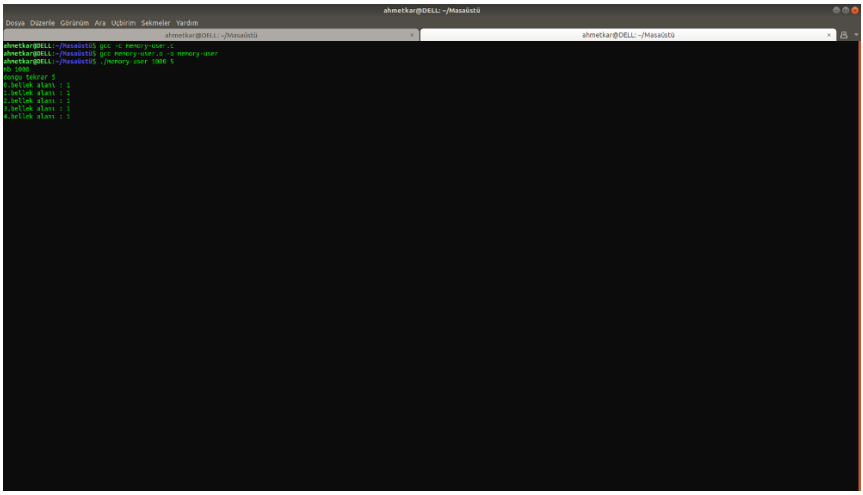
İlk program çalışırken ps auxw komutu aracılığıyla elde ettiğimiz PID si aşağıdaki ekran görüntüsündeki gibidir.

```
root        6483  0.2  0.0      0     0 ?        I    18:18   0:00 [kworker/6:2-mm_]
root        6504  0.0  0.0      0     0 ?        I    18:19   0:00 [kworker/7:3-mm_]
ahmetkar    6511 97.0  0.0 209316   804 pts/2    R+   18:20   0:08 ./memory-user 200 5
ahmetkar    6514  0.2  0.0  23968  4896 pts/0    Ss   18:20   0:00 bash
ahmetkar    6522  0.0  0.0  38800  3464 pts/0    R+   18:20   0:00 ps auxw
```

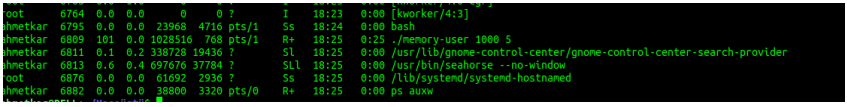
Pmap i "-X" parametresi ve memory user programının PID i ile çalıştırdığımızda sonuç aşağıdaki ekran görüntüsündeki gibidir.Heap in bellek alanı programımızda ayırdığımız bellek alanı kadar artmıştır.



Memory-user programını ilk olarak 100 mb 5 eleman girdileriyle çalıştırdık ve ekran görüntüsü aşağıdaki gibidir.



Program çalışırken ps auxw komutu aracılığıyla elde ettiğimiz PID si aşağıdaki ekran görüntüsündeki gibidir.



Pmap'I "- X" parametresi ve memory-user programının PID i ile çalıştırdığımızda sonuç aşağıdaki ekran görüntüsündeki gibidir.Yine heap in bellek alanı programımızda ayırdığımız bellek alanı kadar artmıştır.