13

Soyutlama: Adres Alanları

Eskiden bilgisayar sistemleri oluşturmak kolaydı. Nedenini biliyor musun? Çünkü kullanıcılar fazla bir şey beklemiyordu. Tüm bu baş ağrılarına "kullanım kolaylığı", "yüksek performans", "güvenilirlik" vb. beklentileri olan kullanıcılar neden olmuştur. Bir sonraki seferde bu bilgisayar kullanıcılarından biriyle tanıştığınızda, neden oldukları tüm sorunlar için onlara teşekkür edebilirsiniz.

## İlkel Sistemler

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

İşletim Sistemi, bellekte bulunan (bu örnekte fiziksel adres 0'dan başlayarak) bir dizi rutin şeklindeydi (aslında bir kitaplık) ve şu an da fiziksel bellekte (başlangıçta başlayan) çalışan bir program (bir işlem) olacaktı. (Bu örnek de 64k fiziksel adresinde) ve belleğin geri kalanını kullandı. Burada çok az yanılsama vardı ve kullanıcı işletim sisteminden pek bir şey beklemiyordu. O günlerde işletim sistemi geliştiricileri için hayat kesinlikle kolaydı, değil mi?

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 Giriş/Çıkış işlemi gerçekleştirmeye karar verildiğinde işletim sisteminin bunlar arasında geçiş yaptığı **çoklu proglamlama** (**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 Mac'inizin pahalı olduğunu düşündüğünüz!) günlerde bile 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 [CV65] üzerinde, toplu hesaplamanın sınırlamalarını fark etti.

1

0KB

Mevcut Program

(kod, veri, vb.)

İşletim Sistemleri (kod, veri, vb.)

64KB

max

Şekil 13.1: **İşletim Sistemleri: İlkel Dönem**

Pek çok kullanıcı aynı anda bir makineyi kullanıyor olabileceğinden ve her biri yürütmekte oldukları görevlerden zamanında bir yanıt beklediğinden (veya umduğundan) **etkileşim(interactivity)** kavramı önemli hale gelmişti. Zaman paylaşımını uygulamanın bir yolu, bir işlemi kısa bir süre çalıştırarak tüm belleğe tam erişim vermek (Şekil 13.1), ardından durdurmak, tüm durumunu bir tür diske kaydetmek (tüm fiziksel bellek dahil olabilir). Başka bir işlemin durumunu yükleyin, bir süre çalıştırın ve böylece uygulayın. Makinenin ham olan bir tür paylaşımı olarak [M+63] gösterilebilir.

Ne yazık ki, bu yaklaşımın büyük bir sorunu var: özellikle hafıza arttıkça çok yavaş kalıyor. Kayıt düzeyinde durumun (Bilgisayar, genel amaçlı yazmaçlar, vb.) kaydedilmesi ve geri yüklenmesi nispeten hızlı olsa da, belleğin tüm içeriğini diske kaydetmek performansızdır. Bu nedenle, işlemleri aralarında geçiş yaparken bellekte bırakmak, işletim sisteminin zaman paylaşımını verimli bir şekilde uygulamasına izin verir ve (Sayfa 3, Şekil 13.2’de gösterildiği gibi) yapmayı tercih ederiz.

Diyagramda üç işlem (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 süreçlerden birini (örneğin A) çalıştırmayı seçerken, diğerleri (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; okumak için bir süreç olmasını istemiyorsanız veya daha da kötüsü olmaması için, başka bir işlemin hafızasını yazın.

0KB

|  |
| --- |
| İşletim Sistemi (kod, veri, vb.) |
| (boş alan) |
| İşlem C (kod,veri,vb.) |
| İşlem B (kod,veri,vb.) |
| (boş alan) |
| İşlem A (kod,veri,vb.) |
| (boş alan) |
| (boş alan) |

64KB

128KB

192KB

256KB

320KB

384KB

448KB

512KB

Şekil 13.2: **ÜÇ İşlem: Belleği Paylaşmak**

## Adres Yolu

Ancak, bu can sıkıcı kullanıcıları aklımızda tutmalıyız ve bunu yapmak için işletim sisteminin **kullanımı kolay(easy to use)** bir fiziksel bellek soyutlaması oluşturması gerekir. Bu soyutlamaya **adres alanı(address space)** diyoruz ve çalışan programın sistemdeki belleğe yansımasıdır. Belleğin temel soyutlamasını anlamak, belleğin nasıl sanallaştırıldığını anlamanın anahtarıdır. Bir işlemin adres alanı, çalışan programın tüm bellek durumunu içerir. Örneğin, programın **kodu(code)** (talimatlar) bellekte bir yerde bulunmak zorundadır ve bu nedenle adres alanındadır. Program çalışırken, işlev çağrı zincirinde nerede olduğunu takip etmek ve ayrıca yerel değişkenleri tahsis etmek, rutinlere ve rutinlerden parametreleri ve dönüş değerlerini iletmek için bir **yığın(stack)** kullanır. Son olarak, **öbek(heap)** dinamik olarak ayrılmış, kullanıcı tarafından yönetilen bellek için kullanılır; örneğin, C'deki bir malloc() çağrısından veya C++ ya da Java gibi nesne yönelimli bir dilde new komutu ile alabileceğiniz gibi orada başka şeyler de var (örneğin, statik olarak başlatılmış değişkenler), ama şimdilik sadece

bu üç bileşeni varsayalım: kod, yığın, öbek. Şekil 13.3'teki örnekte (sayfa 4), küçük bir adres alanımız var (yalnızca 16 KB)1.

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

0KB

1KB

2KB

15KB

16KB

Kod segmenti: talimantın nerede olduğunu gösterir

Öbek segmenti: malloc'd'nin data dinamik veri yapılarını içerir (pozitif olarak büyür)

(boş)

Yığın

Öbek

Program Kodu

(negatif olarak büyür) Yığın segmenti: rutinler için yerel değişken argümanları içerir, dönüş değerleri,vb.

Şekil 13.3: **Adres alanının bir örneği**

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

Sonrasında, program çalışırken büyüyebilen (ve küçülebilen) adres alanının iki bölgesine sahibiz. Öbek en üstte and yığın da altta olacak şekilde yerleşmelidir. 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 ( kullanıcı malloc() aracılığıyla daha fazla bellek istediğinde); yığın 16 KB'de başlar ve yukarı doğru büyür (bir kullanıcı prosedür çağrısı yaptığında söyler).

Ancak, yığının ve öbeğin bu yerleşimi yalnızca bir kuraldır; isterseniz adres alanını farklı bir şekilde düzenleyebilirsiniz (daha sonra göreceğimiz gibi, bir adres alanında birden fazla **iş parçacığı(threads)** bir arada bulunduğunda, adres alanını bu şekilde bölmenin ne yazık ki iyi bir yolu yok).

Elbette adres alanını tarif ettiğimizde, tarif ettiğimiz şey işletim sisteminin çalışan programa sağladığı **soyutlamadır(abstraction)**. Program, 0 ile 16KB arasındaki fiziksel adreslerde gerçekten bellekte 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.

En önemli nokta: Bellek nasıl sanallaştırılır?

İşletim sistemi, birden çok çalışan işlem (tümü paylaşım belleği) için tek bir fiziksel belleğin üzerinde özel, potansiyel olarak büyük bir adres alanının bu soyutlamasını nasıl oluşturabilir?

Sorun da burada başlıyor.İşletim sistemi bunu yaptığında,işletim sisteminin **belleği sanallaştırdığını(virtualizing memory)** söylüyoruz, çünkü çalışan program belleğe belirli bir adreste(0 diyelim) yüklendiğini düşünüyor ve potansiyel olarak çok büyük bir adres alanına sahip(32bit veya 64-bit diyelim); fakat gerçeklik oldukça farklıdır.

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

## Hedefler

Böylece, bu not dizisinde işletim sisteminin yaptığı işe geliyoruz: belleği sanallaştırmak. Ancak işletim sistemi yalnızca belleği sanallaştırmaz. İşletim sisteminin bunu yaptığından emin olmak için bize rehberlik edecek bazı hedeflere ihtiyacımız vardır. Bu hedefleri daha önce gördük (Girişteki cümleri düşünün) ve tekrar göreceğiz, ancak tekrar etmeye değer olduğu için tekrardan söylemede sakınca yoktur.

Bir sanal bellek(SB) sisteminin ana 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 çoğullamak için tüm işi yapar ve bu nedenle yanılsamayı gerçekleştirir.

Sanal makinenin bir diğer amacı da **verimliliktir(efficiency)**. İşletim Sistemi, hem zaman (yani, programları çok daha yavaş çalıştırmamak için) hem de alan (yani, sanallaştırmayı desteklemek için gereken yapılar için çok fazla bellek kullanmaması lazım) açısından sanallaştırmayı mümkün olduğunca **verimli(efficient)** hale getirmeye çalışmalıdır. Zaman açısından verimli sanallaştırmayı uygularken, İşletim Sistemi, Çeviri görünümlü arabellek gibi (ileride öğreneceğimiz) donanım özellikleri de dahil olmak üzere donanım desteğine güvenmek zorunda kalacaktır.

2Bu şeffaflık kullanımı bazen kafa karıştırıcıdır; bazı öğrenciler "şeffaf olmanın" her şeyi açıkta tutmak anlamına geldiğini düşünüyor. Burada bunun tam tersi: İşletim sistemi tarafından sağlanan yanılsamanın uygulamalar tarafından görülmemesi gerektiği anlamına gelir.

Son olarak, sanal makinenin üçüncü hedefi korumadır. İşletim sistemi, 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 erişmemeli veya bunları hiçbir şekilde etkilememelidir. Böylece koruma, süreçler arasında **yalıtım(isolation)** özelliğini sunmamızı sağlar; her süreç, diğer hatalı ve hatta kötü niyetli süreçlerin tahribatından uzak bir şekilde çalışmalıdır.

İpucu: İzolasyon Prensibi

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 tek parça şeklindeki çekirdek tasarımlarından daha fazla güvenilirlik sağlayabilir.

Sonraki bölümlerde, donanım ve işletim sistemleri desteği de dahil olmak üzere, belleği sanallaştırmak için gereken temel **mekanizmalara(mechanisms)** odaklanacağız. Ayrıca, boş alanın nasıl yönetileceği ve alanınız azaldığında hangi sayfaların bellekten atılacağı da dahil olmak üzere, işletim sistemlerinde karşılaşacağınız daha ilgili **politikalardan(policies)** bazılarını da araştıracağız. Bunu yaparken, modern bir

sanal bellek sistemi gerçekten nasıl çalışıyor bunu anlayacağız3.

## Özet

Büyük bir işletim sisteminin alt sisteminin tanıtıldığını gördük: buna sanal bellek deriz. SB 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 böyle devam edeceğiz!

3Ya da sizi kursu bırakmaya ikna edeceğiz. Ama bekleyin; sanal makine aracılığıyla yaparsanız, büyük ihtimalle bu işin sonuna kadar gidersiniz!

Ayrıca: Gördüğünüz her adres sanaldır

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, eğer yazdırabilirseniz, bu aynı zamanda bir sanal adres olacaktır. 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; sadece işletim sistemi (ve donanım) gerçek gerçeği bilir.

İşte main() rutininin konumlarını (kodun olduğu yer), malloc()'tan döndürülen öbeği tahsisli bir değerin değerini ve yığındaki bir tam sayının 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 bitlik Mac'te çalıştırdığımızda aşağıdaki çıktıyı alırız:

kodun yeri:0x1095afe50

öbeğin yeri:0x1096008c0

yığının yeri:0x7fff691aea64

Buradan, kodun önce adres alanından, sonra öbekten geldiğinin ve yığının bu büyük sanal alanın diğer ucunda olduğunu görebilirsiniz. Bu adreslerin tümü sanaldır ve değerleri gerçek fiziksel konumlarından almak için işletim sistemi ve donanım tarafından çevrilmesi gerekir.

# References

[BH70] “The Nucleus of a Multiprogramming System” by Per Brinch Hansen. Communica- tions 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/*](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.](http://www-formal.stanford.edu/jmc/history/timesharing/timesharing.html) *A terrific his- torical 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.*

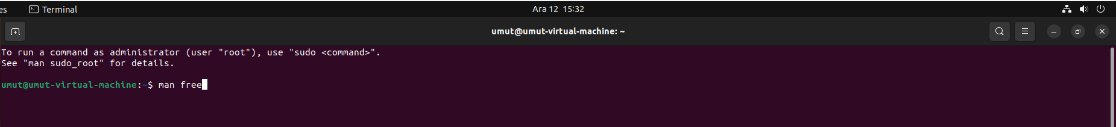
[S+03] “Improving the Reliability of Commodity Operating Systems” by M. M. Swift, B. N. Bershad, H. M. Levy. SOSP ’03. *The first paper to show how microkernel-like thinking can improve operating system reliability.*

# Ödev (Kod)

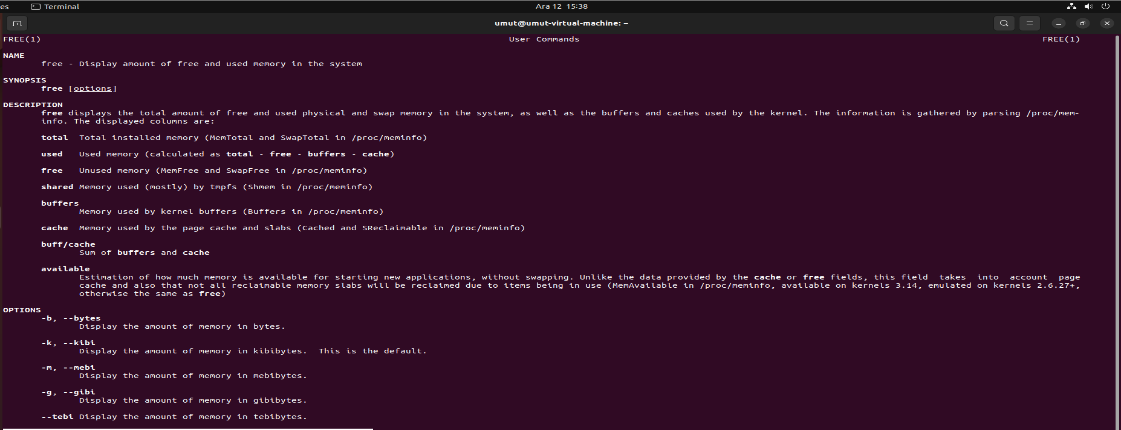
Bu ödevde, Linux tabanlı sistemlerde sanal bellek kullanımını incelemek için birkaç yararlı araç hakkında bilgi edineceğiz. Bu, neyin mümkün olduğuna dair yalnızca özet niteliğinde ipucu olacaktır; gerçekten bir uzman olmak için kendi başınıza daha derine dalmanız gerekecek (her zaman olduğu gibi!).

**Sorular**

* + 1. Kontrol etmeniz gereken ilk Linux aracı olan free’dir ve çok basit bir araçtır. İlk olarak, man free yazın ve kılavuz sayfasının tamamını okuyun; kısa bir yazıdır merak etmeyin!



İlk olarak free’nin ne anlama geldiğini araştırdım. Free komutunun, bir sistemin kullanılan ve kullanılmayan belleğin ve takas belleği(swap) hakkında bilgi verdiğini ve varsayılan olarak, belleği kb (kilobayt) cinsinden görüntülediğini, bellek esas olarak rastgele erişim belleği(RAM) ve takas belleğinden oluştuğunu öğrendim. Terminali açtım ve man free komutunu yazdım yukarıda gösterdiğim gibi. Yazdıktan sonra aşağıda ekran görüntülerini paylaştığım yazılar çıktı. Hepsini internette araştırdım sonra anlamaya çalışarak okudum.



Man Free’nin Manual Free’nin kısaltması olarak yazıldığını öğrendim. Yani kullanıcı özellikleri bir nevi kılavuz sayfası karşıma çıktı. İlk olarak isim başlığı çıktı altında da free- Sistemdeki boş ve kullanılan bellek miktarını görüntüleyin açıklamasını gördüm.

Onun altında sinopsis(yani kısa bir özet) bloğu karşıma çıktı.

Bir sonraki blok Tanım bloğuydu. Altındaki açıklamada şu yazıyordu: free, çekirdek tarafından kullanılan arabellekler ve önbelleklerin yanı sıra sistemdeki toplam boş ve kullanılan fiziksel ve takas belleği(swap) miktarını görüntüler. Bilgiler /proc/meminfo ayrıştırılarak toplanır. Görüntülenen sütunlar da altında yazılmış durumdaydı.

Toplam(total): Toplam yüklü bellek miktarını veriyor.

Kullanılan(used): Kullanılan bellek miktarını veriyor.

Boş alan(free): Kullanılmayan bellek miktarını veriyor.

Paylaşılmış(shared): Tmpfs tarafından kullanılan bellek anlamına geliyormuş. Tmpfs’nin ne olduğunu araştırdığımda ise, işletim sisteminde uygulanan geçici bir dosya depolama olduğunu ve verilerin kalıcı bir depolama aygıtı yerine geçici bellekte depolandığını öğrendim.

Arabellekler(buffers): Çekirdek(kernel) arabellekleri tarafından kullanılan bellektir.

Önbellek(cache): Sayfa önbelleği ve tabakalar tarafından kullanılan bellektir.

Arabellek/Önbellek(buff/cache): Arabellek ve önbelleğin toplamıdır.

Mevcut(available): Değiştirmeden yeni uygulamaları başlatmak için ne kadar bellek bulunduğunun tahminidir. Önbellek veya boş alanlar tarafından sağlanan verilerin aksine, bu alan sayfa önbelleğini ve ayrıca kullanımda olan öğeler nedeniyle tüm geri alınabilir bellek tabakalarının geri kazanılmayacağını dikkate alır.

Bu bilgilerinin altında özellikler adında bir blok gördüm. Burada aslında bellek boyutunun farklı türlerde göstermenin bizim elimizde olduğunu öğrendim.

free --bytes yazarsak byte şeklinde

free --kibi yazarsak kibibyte şeklinde (Hiçbir değişiklik yapmazsak bizim bellek değerlerimiz kibibyte şeklinde görünür)

free --mebi yazarsak mebibyte şeklinde

free --gibi yazarsak gibibyte şeklinde

free --tebi yazarsak tebibyte şeklinde

free --pebi yazarsak pebibyte şeklinde

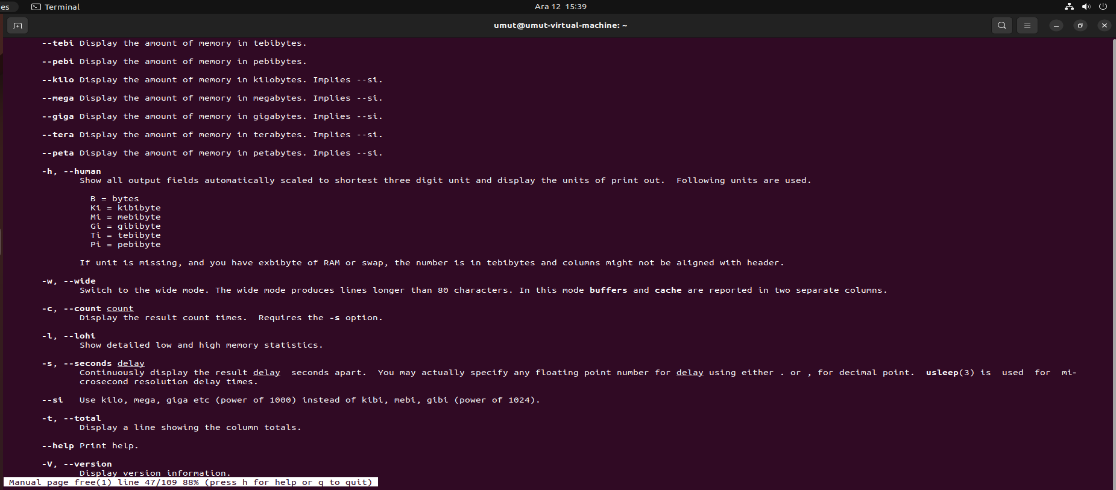
free --kilo yazarsak tebibyte şeklinde

free --mega yazarsak megabyte şeklinde

free --giga yazarsak gigabyte şeklinde

free --tera yazarsak terabyte şeklinde

free --peta yazarsak petabyte şeklinde



-h, --human komutla da, Tüm çıktıları otomatik olarak en kısa üç basamaklı birime ölçeklenmiş olarak gösterir ve çıktı birimlerini görüntüler. Aşağıdaki birimleri kullanırız.

B = bytes

Ki = kibibyte

Mi = mebibyte

Gi = gibibyte

Ti = tebibyte

Pi = pebibyte

-w, --wide işlemiyle, Geniş moda geçirir. Geniş mod, daha uzun çizgiler üretir

(80 karakterden fazla). Bu modda arabellekler ve önbellek iki ayrı sütunda raporlanır.

-c, --count count işlemi, sonuç sayım sürelerini gösterir.

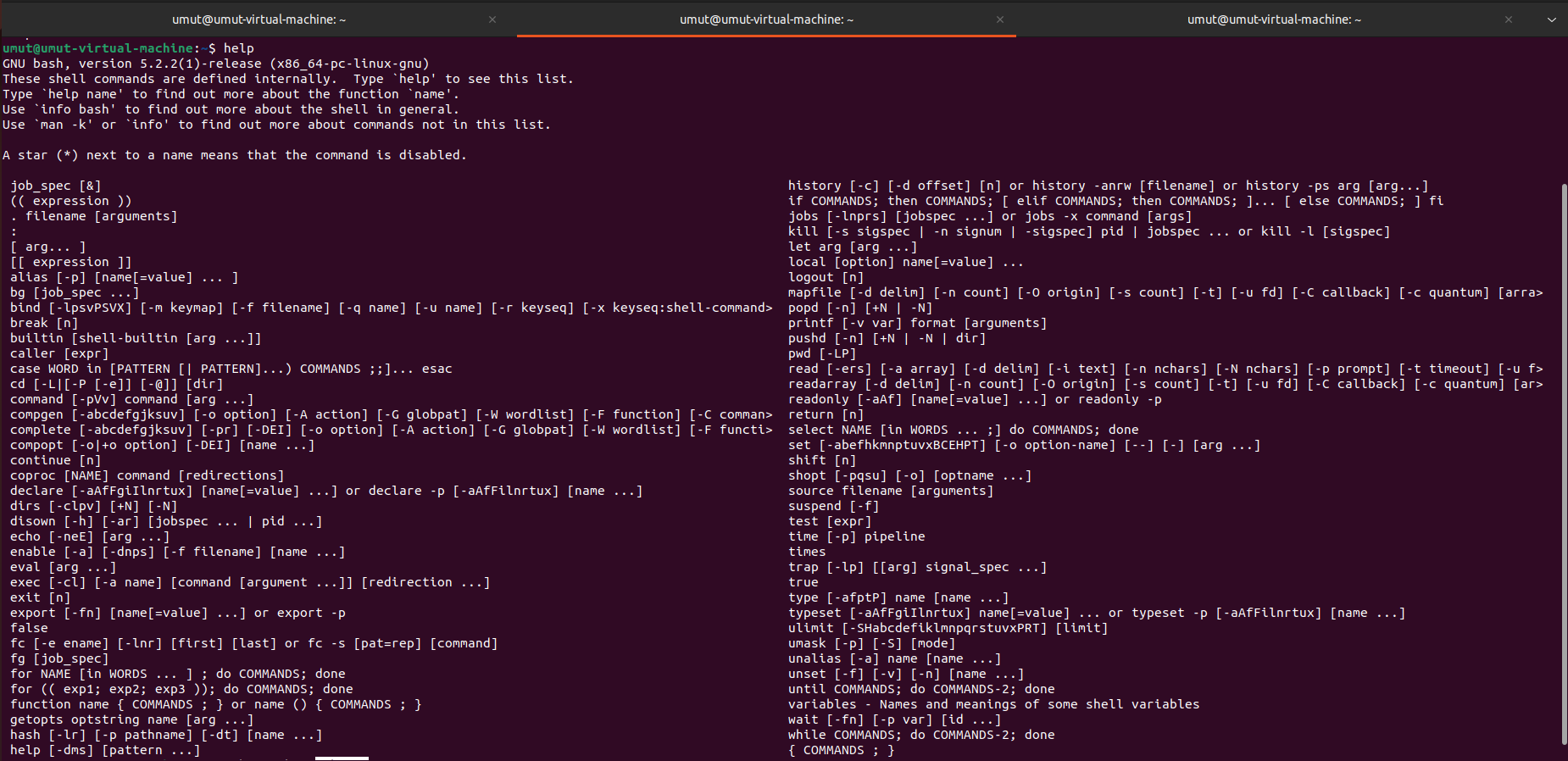
-l, --lohi işlemi, düşük ve yüksek bellek istatistiklerini ayrıntılı olarak gösterir.

-s, --seconds delay işlemi, sonuç gecikmesini saniyeler arayla sürekli olarak görüntülebiliriz. Gecikme için herhangi bir kayan nokta sayısı belirtebiliriz. veya , ondalık nokta için kullanırız.

--si işlemi, kibi,mebi,gibi(1024’ün katı) değerler yerine kilo,mega,giga(1000’in katı) değerler kullanmak için kullanılır.

-t, --total işlemi, Kolon toplamlarının sayısını verir.

--help işlemiyle terminal hakkında daha geniş bilgiye sahip olabiliriz. Kısaltmalarla alakalı bilgileri öğrenebiliriz. Ben yazdığımda bu satırlar çıktı.

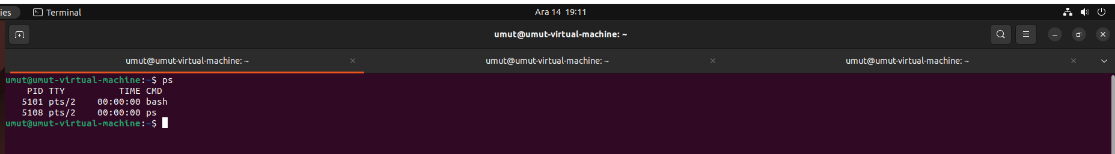




-V, --version işlemi, sürüm bilgilerini ekranda gösterir.

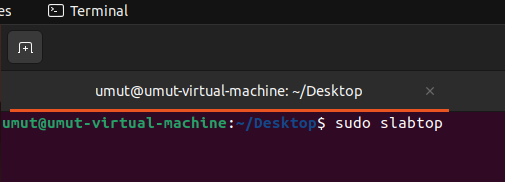
Böylelikle özellikler bloğu bitmiş oldu. En altta da Dosyalar, hatalar yeri var. Dosyalarda hafıza bilgisinin yeri görünüyor ve hata alırsak o hatayı göndereceğimiz bir web sitesi linki mevcut.

Ayrıca bakınız yerinde yazan ps(1), slabtop(1), top(1), vmstat(8) ifadelerini de yazdım.



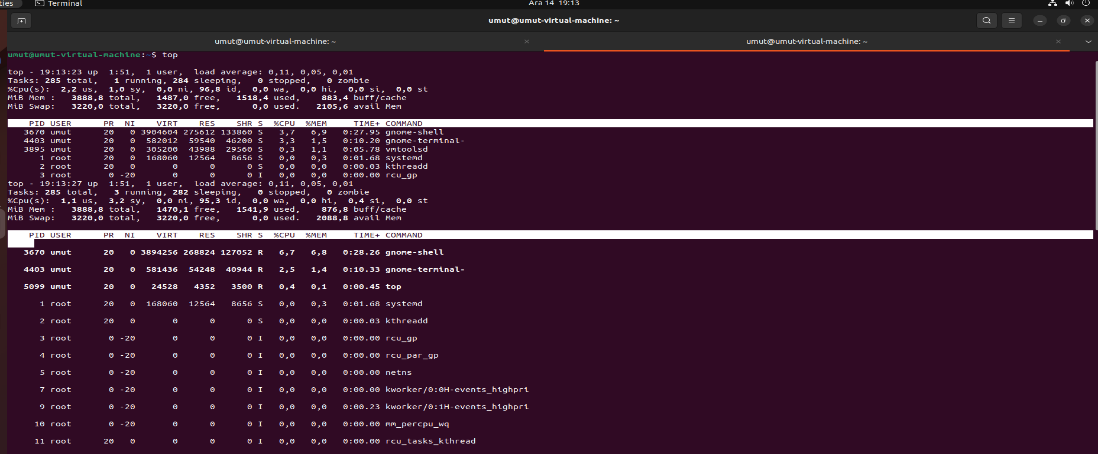
ps (process status) Şimdi çalışan işlemleri gösterir.

Burada süreç numarası(process id) ve teletypewriter(tty) ve CMD bilgileri çıktı.

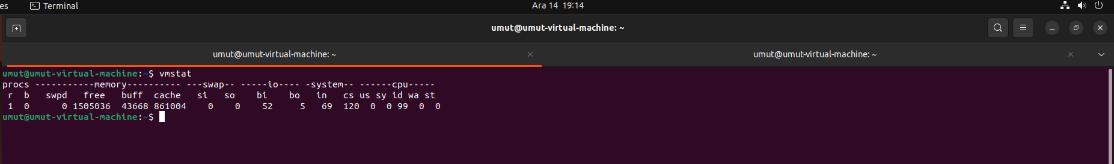




Slabtop, gerçek zamanlı olarak ayrıntılı çekirdek tabakasının önbellek bilgilerini görüntüler. Listelenen sıralama ölçütlerinden birine göre sıralanmış en iyi önbelleklerin bir listesini görüntüler. Ayrıca döşeme katmanı bilgileriyle dolu yukarıda gösterilen istatistik başlığını gösterir.

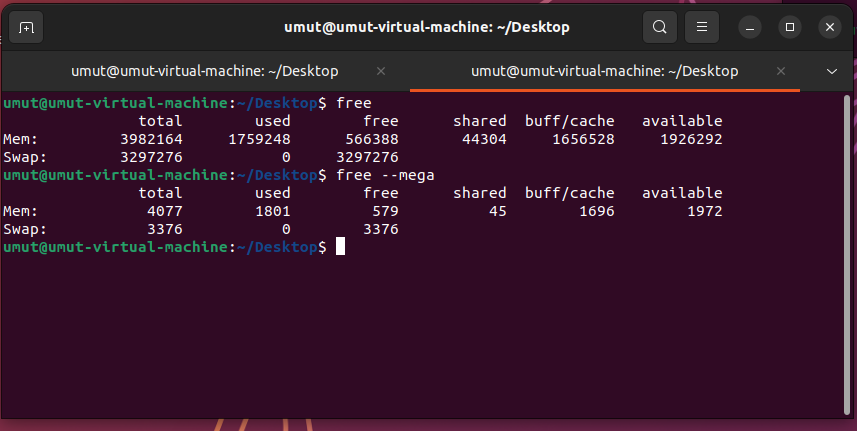


Linux işlemlerini göstermek için top komutu kullanılır. Çalışan sistemin dinamik, gerçek zamanlı bir görünümünü sağlar. Genellikle, bu komut sistemin özet bilgilerini ve halihazırda Linux Çekirdeği tarafından yönetilen işlemlerin veya iş parçacıklarının(threads) listesini gösterir.



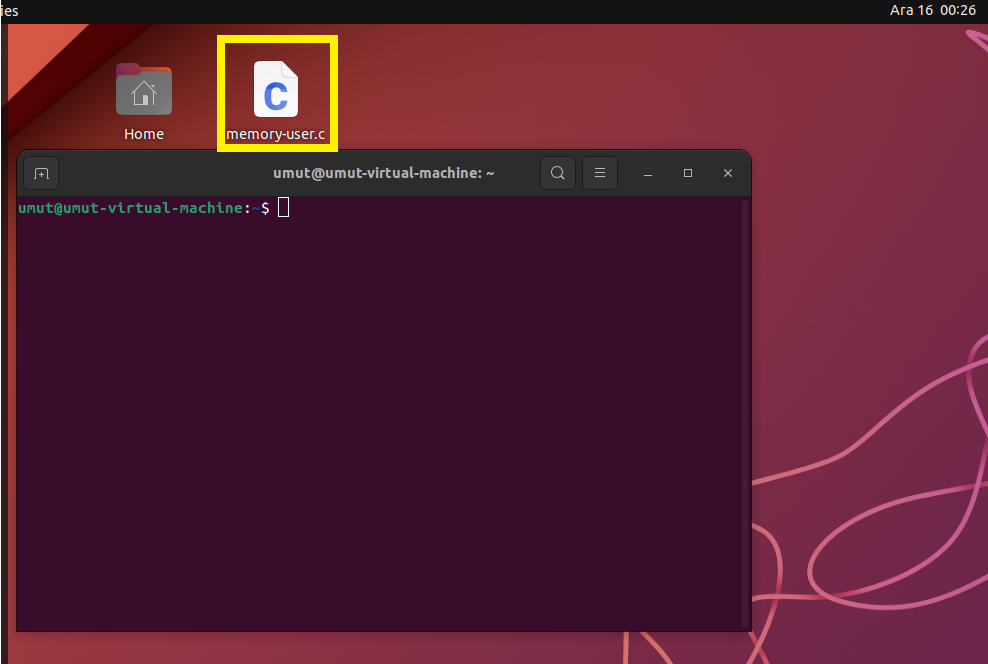
vmstat kodu sanal makinenin özelliklerini gösteriyor. Boş alanı(free), arabelleği(buff), önbelleği(cache) bilgilerini görebiliriz.

2. Şimdi, free komutunu çalıştırın, belki yararlı olabilecek bazı bağımsız değişkenleri kullanarak (örneğin, -m, bellek toplamlarını megabayt olarak görüntülemek için) çalıştırın. Sisteminizde ne kadar bellek var? Belleğin ne kadarı boş alan? Bu sayılar tahminlerinize uyuyor mu?

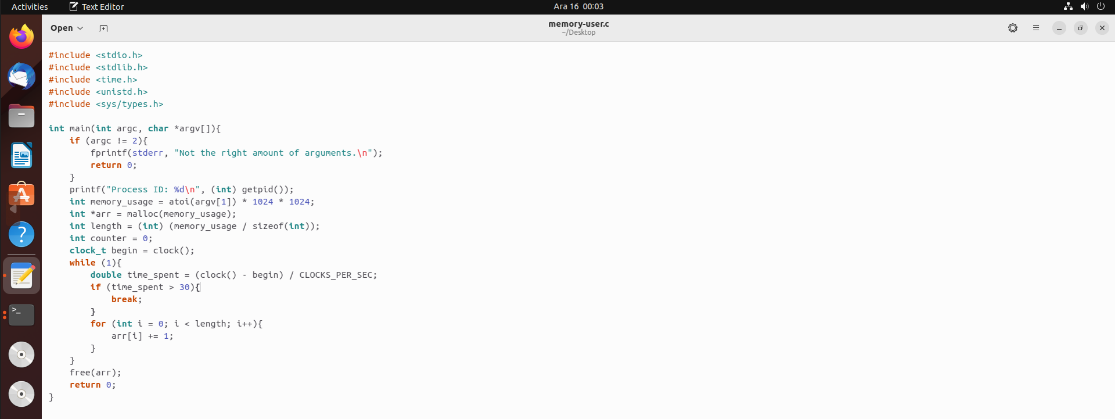


İlk olarak free yazarak bilgilere ulaştım, ama bu veriler kibibyte olarak verilmişti. Sonra free --mega komutu ile verilerin megabyte olarak yazdırılmasını istedim. Yukarıdaki ekran görüntüsünde hem bellek(memory) hem de takas alanı(swap) hakkındaki bilgilere ulaşabiliyoruz. Ben sanal belleğin 4GB olmasını istemiştim(4.096MB) yukarıda hafızanın toplam 4077MB olduğu görünüyor istediğime çok yakın bir değer verdi. Belleğin 1801MB’si kullanılmış durumda. Bellekte 579MB boş alan olduğunu gösteriyor. 45MB paylaşılmış belleğe sahipmişim temporary filesystem(tmpfs)’in kullanabileceği yani kalıcı depolama yerine geçici dosya depolama alanı için kullabileceğim 45MB’m olduğunu öğrendim. Arabellek/Önbellek oranının da 1696MB olduğu ve mevcut olarak kullanılabilecek 1972MB’min daha olduğu bilgisine eriştim. Takas alanıyla(swap) ilgili bilgiler de belleğin hemen altında verilmiş durumda. Toplam 3376MB alanım var. Şu anlık yüksek boyutta bir şey çalışmadığından takas alanına ihtiyaç yoktur. Kullanılan alan 0 olarak görünmesi normal bu yüzden de boş alan tam olarak toplamla aynı MB sayısına sahip.

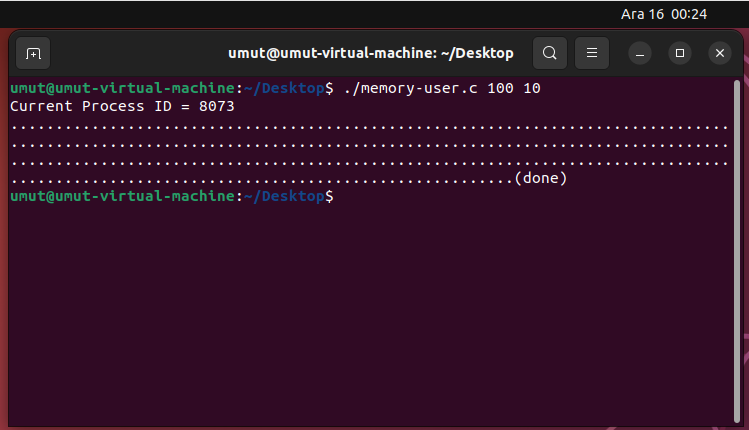
3. Ardından, memory-user.c adlı, belirli bir miktarda bellek kullanan küçük bir program oluşturun. Bu program bir komut satırı argümanı almalıdır:kullanılan megabayt bellek sayısını göstersin. Çalıştırıldığında, bir dizi tahsis etmeli ve her satıra yazarak dizi boyunca sürekli akış yapmalıdır. Program bunu süresiz olarak veya komut satırında belirtilen belirli bir süre boyunca yapmalıdır.



İlk olarak memory-user.c diye metin belgesi oluşturdum ve bunu masaüstüne kaydettim. İçini açıp benden istenen programın kodunu yazdım.

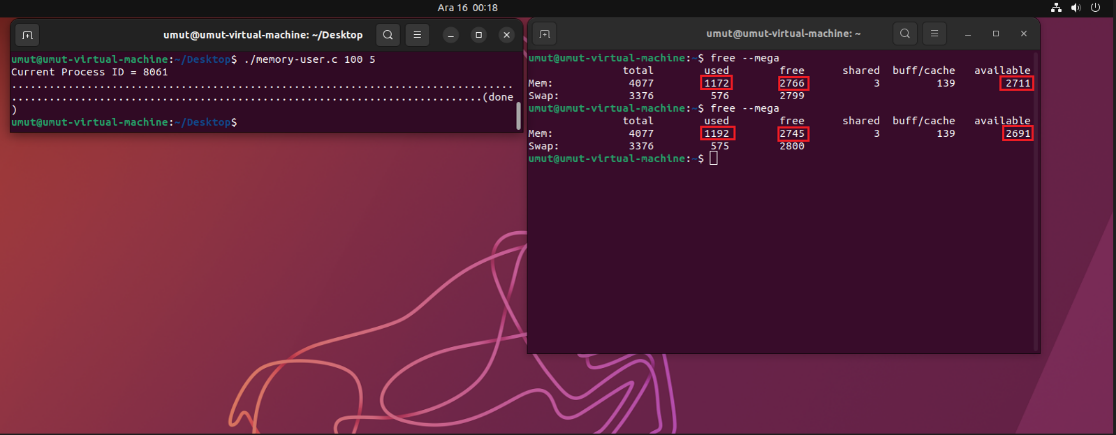


Kodu yazmaya #include<stdio.h> , #include<stdlib.h> , #include<time.h> #include<unistd.h> , #include<sys/types.h> yazarak gerekli kütüphaneleri ekleyerek başladım. Argc parametresi, girilcek parametre sayısını temsil ediyor. Eğer parametre sayısı 2 değilse Doğru sayıda argüment girilmedi hatasını veriyor. 2 değer yazarsak da if bloğuna girmeden koda devam ediyor. Argv(1)’ı memory\_usage’e atadım. Uzunluk diye bir değer oluşturdum ve bu uzunluk denilen ifadenin karşılığı memory\_usage/sizeof olarak yazdım. Counter değerini 0’dan başlattım. Clock değerini başlattım. Sonsuz döngü içinde time\_spent diye bir değişken ekledim ve bunu clock()-begin/Clocks\_per\_sec şeklinde atadım. Eğer süre 30 saniyeyi geçerse kod break olsun diye de bir kod ekledim. Sonra for döngüsüne girerek arr değerini 1 arttırdım ve bu artan değer belleğe yansıması için de free(arr); komutu ile de değişimin free’ye aktarılmasını sağladım. Return komutuyla da kodu bitirdim.

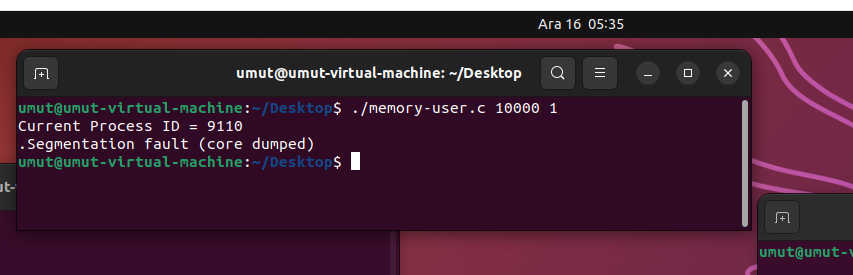


Soruda da belirttiği gibi sorunun çözümü tek parametreyle ya da iki parametleyle yapılabilirdi. Metin belgesinin adı memory-user.c olduğu için belgedeki kodu çalıştırmam gerekiyordu onun için ./memory-user.c şeklinde yazmam gerekiyordu. Ben iki parametreyle yapmayı seçtim yani zaman değerini de kendim ekledim. Yukarıda göründüğü gibi ilk değer (100 olan değer) boyutu ikinci değer de (10) süreyi gösteriyor. Bu kodu yazdığım zaman noktalar soldan başlayarak yazdığım süre boyunca yani 10 saniye dolana kadar ekranda nokta şeklinde yer kapladı.

4. Şimdi, bellek kullanıcı programınızı çalıştırırken, aynı zamanda (farklı bir terminal penceresinde, ancak aynı makinede) boş aracı çalıştırın. Programınız çalışırken bellek kullanım toplamları nasıl değişir? Bellek kullanıcı programını sonlandırdığınızda ne olur? Rakamlar beklentilerinizle örtüşüyor mu? Farklı bellek kullanımı miktarları için bunu deneyin. Gerçekten büyük miktarda bellek kullandığınızda ne olur?

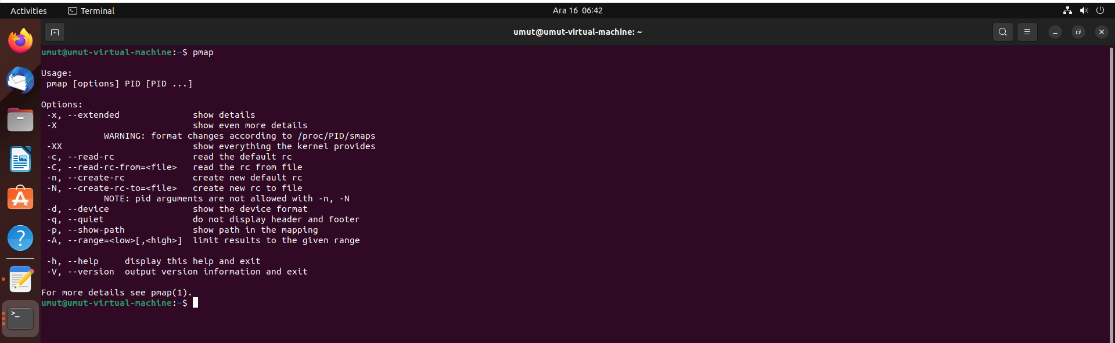


İlk olarak solda kodu yazacağım terminali ve bellek özelliklerini görebileceğim terminali de sağda açtım. Sağdaki terminalde belleğin ilk değerlerini kaydettim sonra ./memory-user.c 100 5 kodu ile 100 değerini denedim ve 5 saniye boyunca ekrana nokta koyarak ilerledi ve yan pencerede tekrardan bellek değerlerinin nasıl değiştiğini görmek için free komutunu yazdım. Bellekte değişen değerler arasında kullanılan bellek miktarı vardı 1172MB iken, 1192MB değerine yükselmişti. Kullanılmayan bellek miktarında da azalış olduğunu gözlemledim 2766MB’den, 2745MB değerine düşmüştü. Mevcut bellek değeri de 2711MB’den 2691MB’ye düşüş göstermişti.



Bellek kullanım miktarını yüksek bir değer yapınca da örneğin burada 10000 değerini seçtim. Segmentation fault (core dumped) hatasını verdi ve bu yüzden bellek özellikleri değişmedi.

5. Pmap olarak bilinen bir araç daha deneyelim. Anlamak için biraz zaman ayırın ve pmap kılavuz sayfasını ayrıntılı olarak okuyun.



Linux'ta pmap komutu, bir işlemin bellek haritasını görüntülemek için kullanılır. Bir hafıza haritası, hafızanın nasıl dağıldığını gösterir. Pmap yazdıktan sonra çıkan komutları yanındaki açıklamalarıyla birlikte okudum.

pmap -x PID komutu ile bilgileri genişletilmiş biçimde görüntüleriz.

pmap -XX PID komutu, çekirdeğin bize sağladığı her türlü bilgiyi görmek için kullanılır.

pmap -c PID komutu, varsayılan yapılandırmayı okumak için kullanılır.

pmap -n komutu, yeni konfirigasyon oluşturmak için kullanılır.

pmap -d PID komutu ile ekstra bilgiler de görünür.

pmap -q PID komutu üst bilgi ve alt bilgi satırını göstermez.

pmap -p PID komutu eşleme sütunundaki dosyaların tam yolunu gösterir.

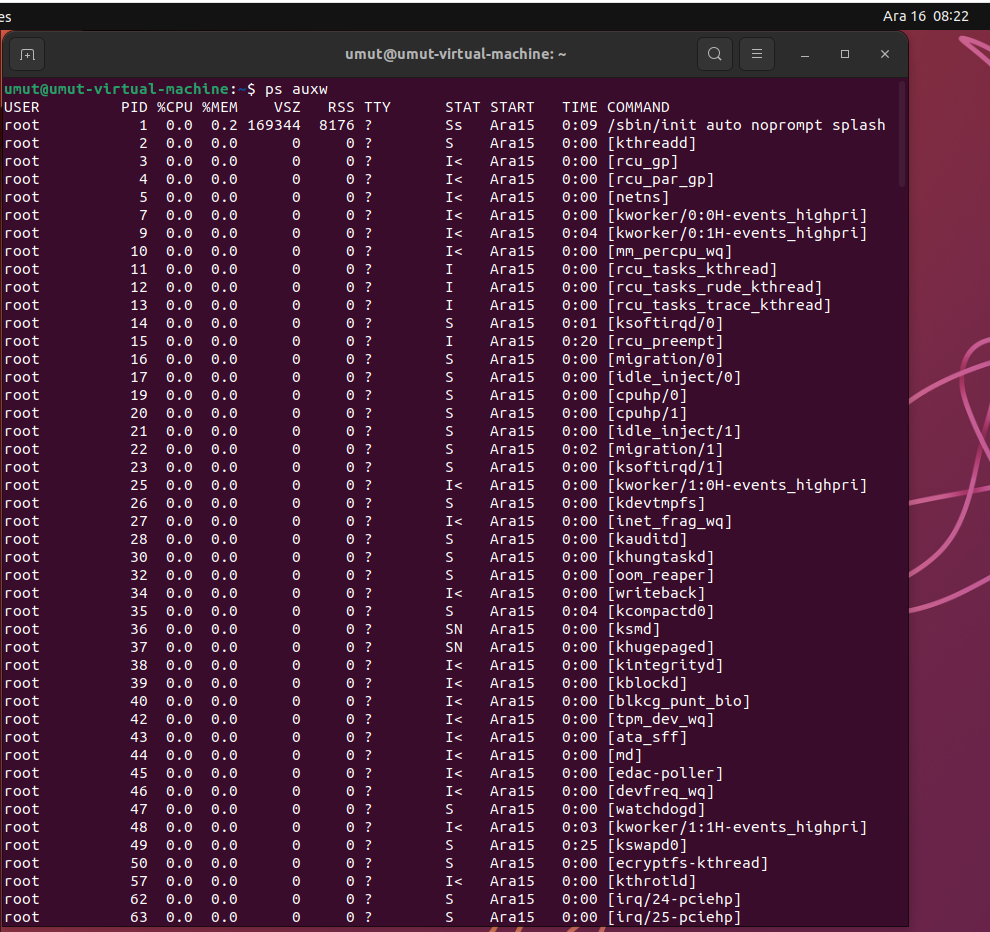
pmap- A komutu ile verilen aralıktaki sonuçları görüntülemek için kullanılır.

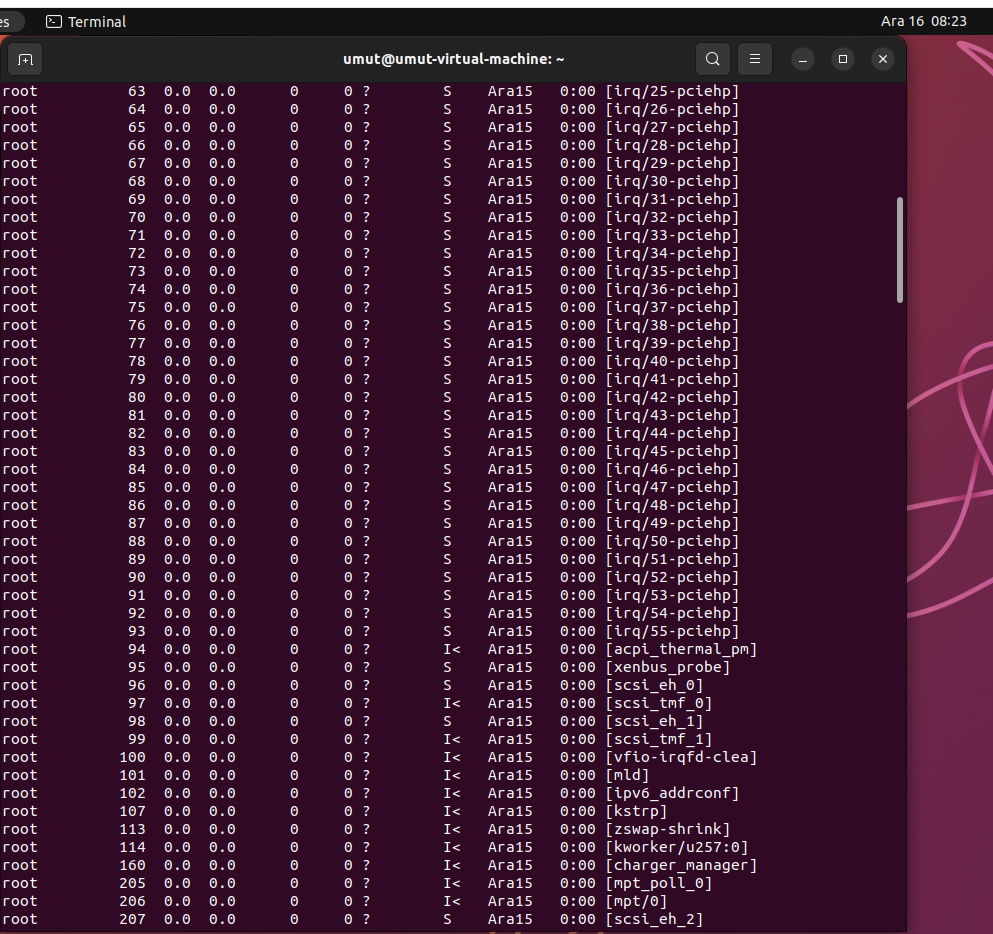
pmap -h komutu, yardım metnini görüntülemek için kullanılır.

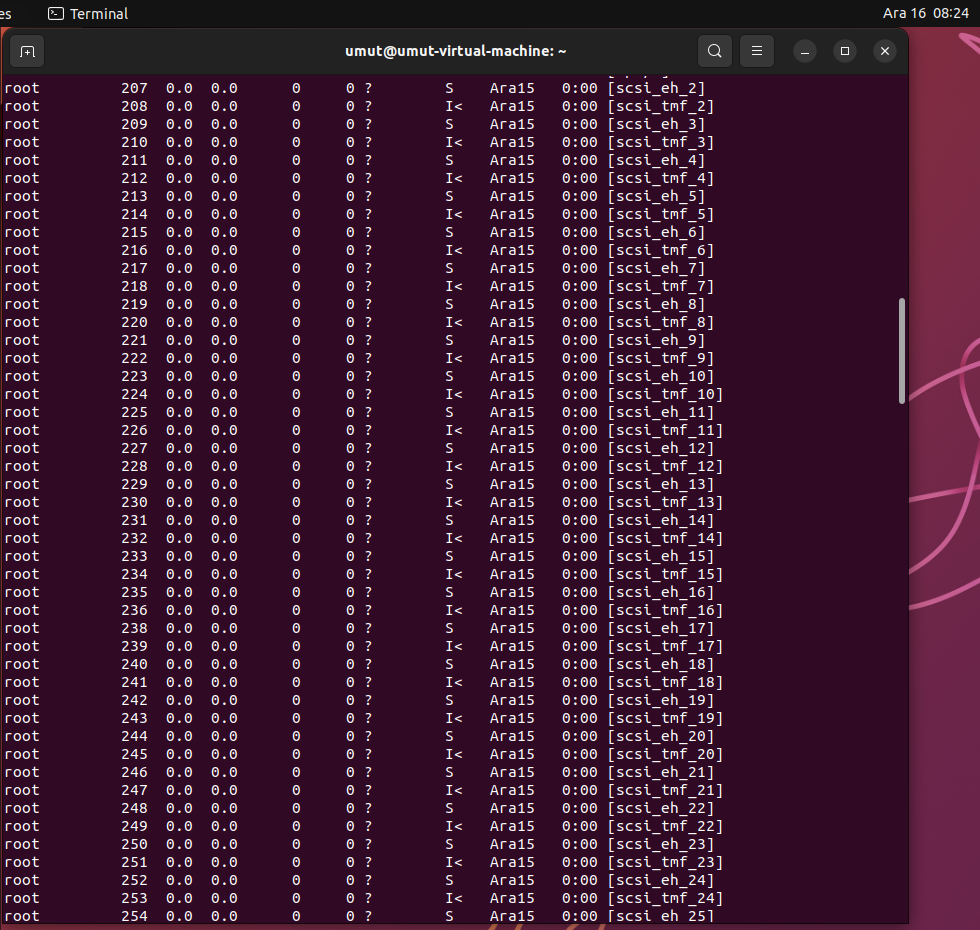
pmap -V komutu sistemin kullandığı pmap sürümünü gösterir.

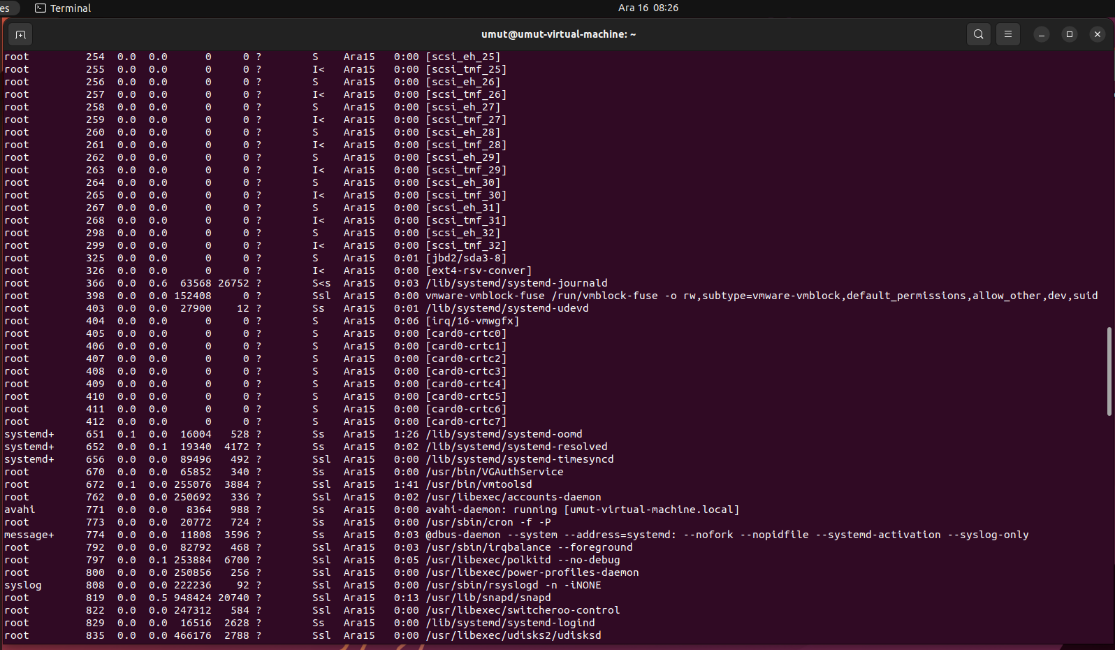
6. Pmap'i kullanmak için, üzerinde çalıştığınız işlemin işlem kimliğini bilmeniz gerekir. Bu nedenle, tüm işlemlerin bir listesini görmek için önce ps auxw'yi çalıştırın; ardından tarayıcı gibi ilginç bir tane seçin. Bu durumda bellek kullanıcı programınızı da kullanabilirsiniz (aslında, bu programın getpid()'i çağırmasını ve size kolaylık sağlamak için PID'ını yazdırmasını bile sağlayabilirsiniz).

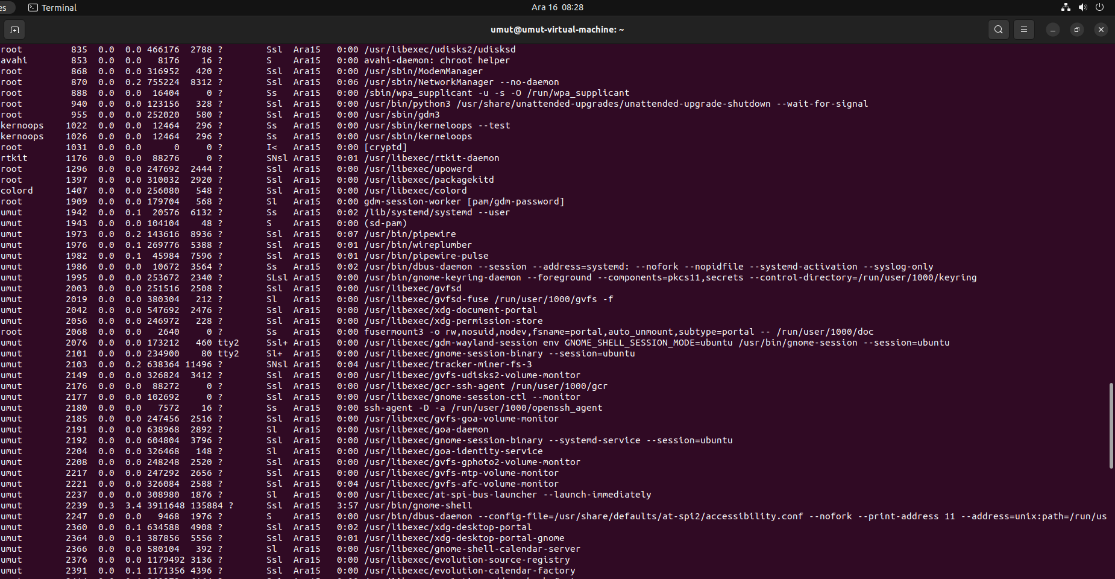
ps auxw komutunu yazdım ve önüme çok uzun ve birçok değişkene sahip olan değerler çıktı. İlk önce çıkanların ekran görüntülerini koydum, açıklamayı altında yapacağım.

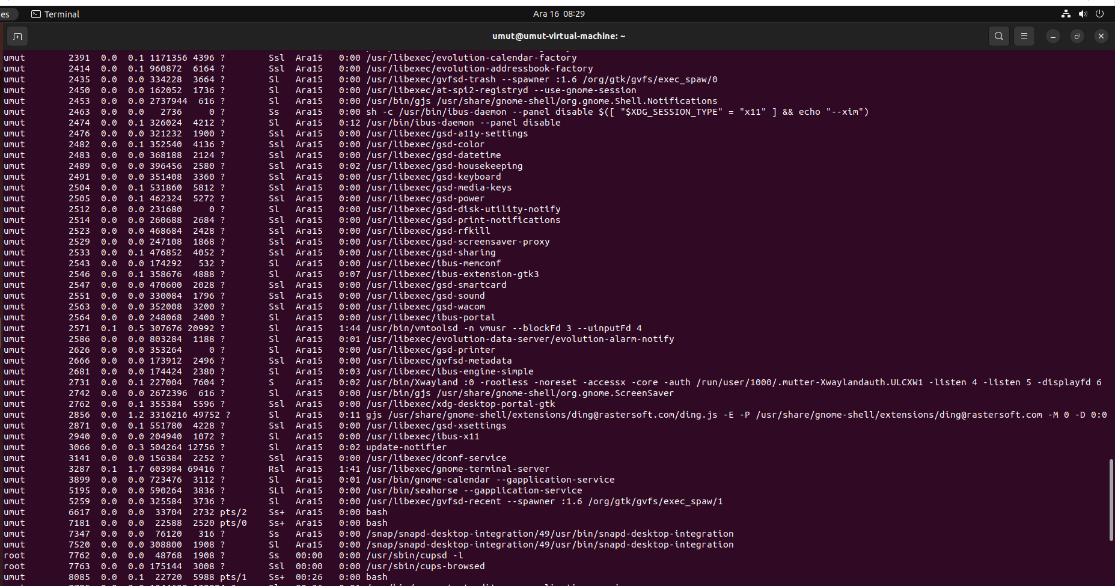


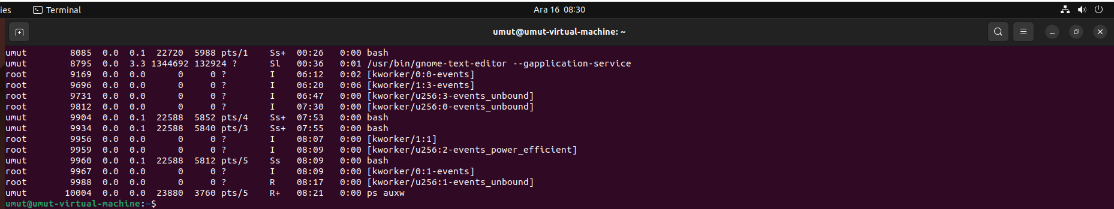


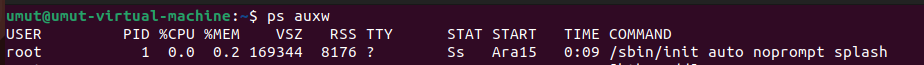












Yukarıda örnek olarak verdiğim satırdaki değerleri açıkladım.

USER yani kullanıcı bilgisini veriyor bu root,umut,systemd+ veya syslog gibi başka kullanıcı olabilir.

PID(Process ID) işlem kimlik numarası yazıyor. PID özeldir ve her işlem için farklı bir değer alır.

%CPU: İşlemciyi yüzdelik olarak ne kadar kullandığı yazıyor.

%MEM: Belleği yüzdelik olarak ne kadar kullandığı yazıyor

VSZ(Virtual Memory Size): Sanal bellek boyutudur.Değiştirilen, tahsis edilen, ancak kullanılmayan ve paylaşılan kütüphanelerdeki bellek de dahil olmak üzere, işlemin erişebileceği tüm bellekleri içerir.

RSS(Resident Set Size): Bu işleme ne kadar bellek ayrıldığını ve RAM’de olduğunu göstermek için kullanılır. Takılan belleği içermez. Bu kütüphanelerdeki sayfalar gerçekte bellekte olduğu sürece, paylaşılan kütüphanelerdeki belleği ve tüm yığın, yığın belleğini içerir.

TTY(TeleTyPewriter): Standart girişe bağlanan dosyanın standart çıkışını yazan bir komuttur.

Stat: Dosya ve dosya sistemi hakkında bilgi veren bir komuttur.

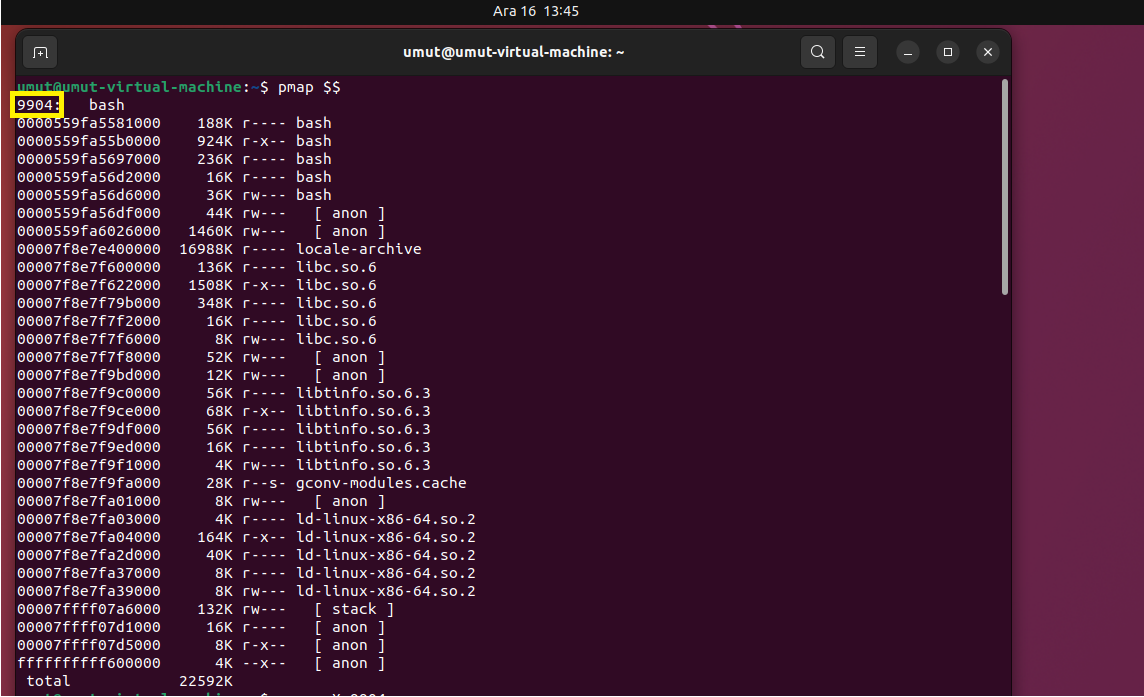
Start: İşlemin ne zaman başladığını belirtir. Örneğin yukarıda Ara15 yazıyor yani aralık 15’te başladığını belirtiyor.

TIME: Kullanıcının CPU kullanım süresidir.

COMMAND: Komut bloğu olarak hangisinde olduğunu gösterir. Örnek olarak yukarıda /sbin/init auto noprompt splash komutunda olduğunu anlıyoruz.

7. Şimdi pmap'i bu işlemlerin bazılarında çeşitli işaretler kullanarak çalıştırın (örneğin-X) işlemle ilgili birçok ayrıntıyı ortaya çıkarmayı sağlar. Bunu yapınca ne gördün? Basit kod/yığın/öbek anlayışımızın aksine, modern bir adres alanını kaç farklı varlık oluşturur?

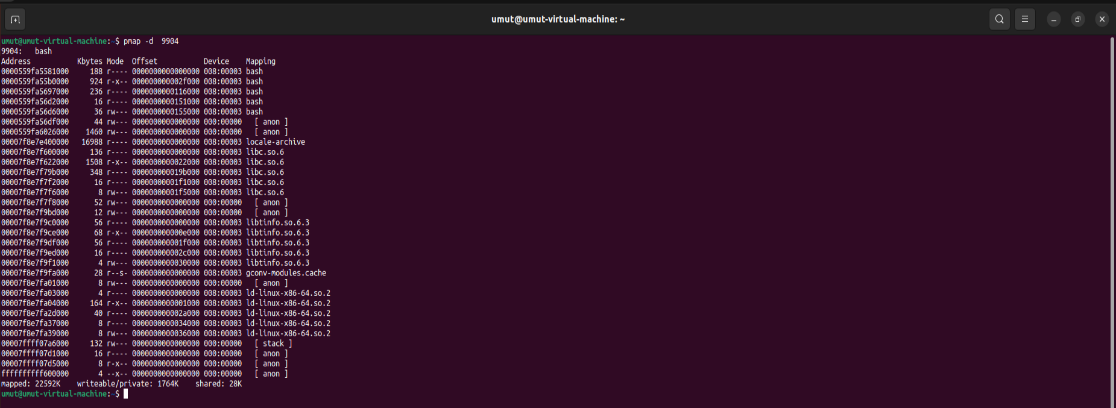
İlk olarak PID değerini bilmemiz lazım onun için de pmap $$ komutunu yazdım ve ekran görüntüsünde gösterdiğim gibi değeri 9904 olarak gördüm.



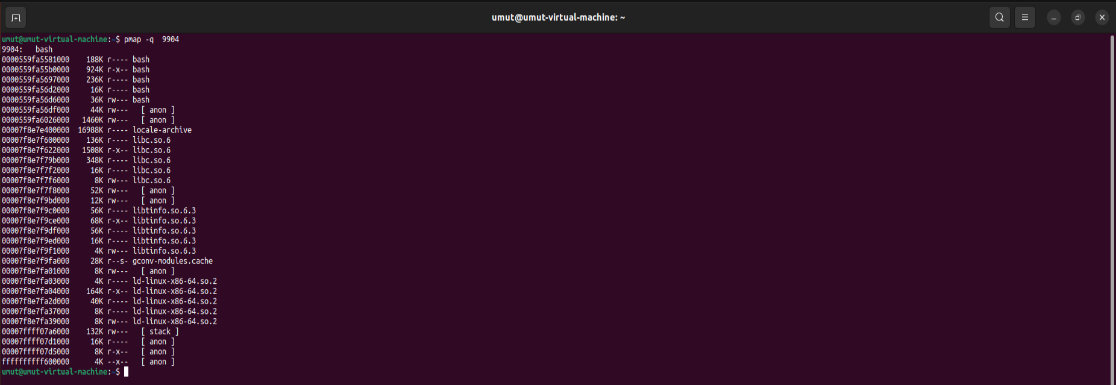
Bu sorunun cevabı için -X, -d ve -q işaretlerini kullanmayı düşündüm.



Pmap -X 9904 komutunu yazınca genişletilmiş bir şekilde görürüz değerleri. Sorudaki “Bunu yapınca ne gördün? Basit kod/yığın/öbek anlayışımızın aksine, modern bir adres alanını kaç farklı varlık oluşturur?” sorusuna da ekranda yazan varlıkları sayınca 21 tane varlık olduğunu gördüm.

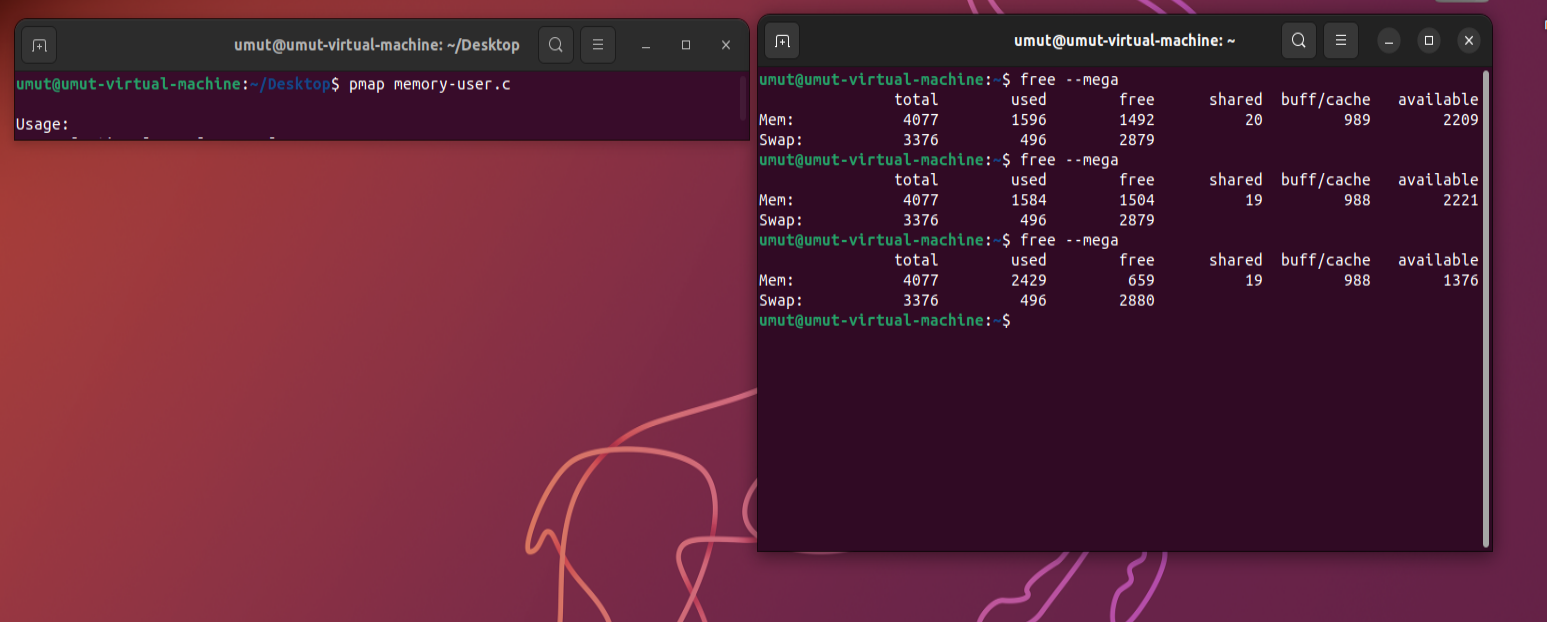


Pmap -d 9904 yazınca, pmap -x 9904’ten farklı olarak Kbytes ve Mode bilgisine de ulaşabildik. Ama -X kadar geniş bir bilgi vermediği için birkaç değer de eksik örneğin: Inode, Size, Rss, Pss, Referenced, Anonymous, LazyFree, ShmemPmdMapped, FilePmdMapped, Shared\_Hugetlb, Private\_Hugetlb, Swap, SwapPss Locked, THPeligible, ProtectionKey gibi değerleri göstermiyor.



Pmap -q 9904 yazdığımızda da Address, Kbytes, Mode ve Maping değerleri çıkıyor ama bu değerlerin üstünde ne oldukları yazmaz. Pmap -d 9904’deki Offset ve Device değerleri de ekranda göstermez.

8. Son olarak, bellek-kullanıcı programınızda farklı miktarlarda kullanılan bellekle pmap'i çalıştıralım. Burada ne görüyorsunuz? pmap çıktısı beklentilerinizi karşılıyor mu?



pmap memory-user.c komutu ile çalıştırmadan önce bellek değerlerimi ekrana yazdırdım. Kullanılan 1584MB’den 2429MB’ye çıktı. Boş alan da 1504MB’den 659MB’ye düştü. En son da Mevcut bellek değeri 2221MB’den 1376MB’ye geldi.