Bölüm: Bellek APİ (uygulama programlama arayüzü)

Bu bölümde , unix sistemlerinde bellek ayırma arayüzleri hakkında konuşacağız. Sağlanan arayüzler oldukça basitler, ve bu yüzden bölüm oldukça kısa ve öz. Asıl sorunumuz şudur:

Bellek nasıl ayrılır ve yönetilir:

Unix ve C programlarında belleğin nasıl ayrılacağını ve yönetileceğini anlamak sağlam ve güvenilir yazılımı oluşturmak için çok kritiktir. Çoğunlukla hangi arayüzler kullanılır ? Hangi hatalardan kaçınmalıyız ?

Bellek türleri

C programını çalıştırırken, ayrılan iki tür bellek vardır. Ilkine **stack (yığın)** bellek denir , ve **ayırıcılar ve yeniden ayırıcılar (allocations , deallocations)** sizin için dolaylı yoldan derleyici tarafından yönetilir, programcı bu sebepten bazen bunu ***otomatik* hafıza (*outomatic memory)*** olarak adlandırır

C'deki yığın üzerinden hafızaya **bildirmek (declaring)** kolaydır. Mesela, **integer** için fonksiyonda **func**() biraz boşluğa ihtiyacımız var diyelim, **integer**'a x atayalım. Bir hafıza parçasına bildirmek için, sadece bunun gibi bir şey yap:

Void func() {

Int x; // yığına bir tamsayı bildirir

func() komutunu çağırdığımızda yığında boşluk açtığımızdan emin olduktan sonra derleyici gerisini halleder. Fonksiyonu geri çevirdiğimizde (**return**) derleyici belleği sizin için birleştirir böylece yürütme çağırısının ötesinde bir bilgi istiyorsan, bu bilgiyi yığına bırakmasan iyi olur.

Uzun ömürlü hafızaya olan ihtiyacımız bizi ikinci türe götürmüştür. Bu türe ***heap*** bellek denir, tüm ayırıcılar ve yeniden ayırıcılar açıkça siz programcılar tarafından işlenir. Şüphesiz, ağır bir sorumluluk ve kesinlikle bir çok hatanın sebebi. Ama dikkatli olursanız bu arayüzleri doğru ve hatasız kullanacaksınız. Burda **heap** bellek üzerinde bir tam sayının nasıl ayrılacağına dair bir örnek:

Void func() {

*Iny *x = (int *) malloc(sizeof(int));*

…

}

Bu küçük kod parçacığı hakkında birkaç not. ilk olarak, bu satırda hem **stack** hem de **heap** ayırmanın gerçekleştiğini farkedebilirsiniz: ilk olarak derleyici, **işaretçiniz(pointer)** bir tam sayı gördüğünde yer açmayı bilir burdaki işaretçinin bildirimi (int *x); daha sonra, program **malloc**() öğesini çağırdığı zaman yığından tam sayı için yer ister; rutin olarak bir tam sayı adresi böyle döndürülür(başarılı yada başarısızlık halinde NULL) bu daha sonra kullanılmak üzere **stackt**e saklanır.

Açık doğası ve daha çeşitli kullanımı nedeniyle heap bellek hem kullanıcılar için hemde sistemler için daha fazla zorluk sunar, bu tartışmamızın kalanının odak noktasıdır

Malloc() çağrısı

Malloc() çağrısı oldukça basittir: **heap**teki oda boyutunu sorma kısmını geçiyorsun ve ya başarılı olur ve işaretçi sana yeni ayrılmış alanı verir yada başarısız olur ve **NULL** döndürür.

Kılavuz sayfası **malloc**'u kullanmanız için size ne yapmanız gerektiğini gösterir; komut satırına man malloc yazın ve bunları göreceksiniz:

#include <stdlib.h>

…

Void *malloc(size_t size);

Bu bilgiye göre, **malloc**'u kullanmak için tüm ihtiyacınız olan başlığa *stdlib.h* dosyasını eklemektir. Aslında, bunu yapmanıza gerçekten gerek yok, **malloc**() kodu C programındaki tüm C kütüphanelerinde var; komutu eklemek sadece **malloc**()'u doğru şekilde çağırıp çağırmadığınızı kontrol eder (örn. Doğru sayıda , doğru türde bağımsız değişken). **Malloc**()'un aldığı tek parametre t olup kaç bayta ihtiyacınız olduğunu basitçe açıklar. Ancak çoğu programcı buraya direkt olarak sayı yazmamalıdır (10 gibi); gerçekten, bunu yapmak kötü bir form olarak düşünülebilir. Bunun yerine çeşitli rutinler ve makrolar kullanılmış

Ipucu: şüphe olduğunda dene

Kullandığınız bazı rutin operatörlerin nasıl davrandığından emin değilseniz, sadece denemek ve düzgün çalıştığından emin olmak hiçbir şeyin yerini tutamaz. Kılavuz sayfaları veya diğer dökümanlar kullanışlıdır, önemli olan pratikte nasıl çalıştığıdır. Biraz kod yaz ve test et! Şüphesiz kodun arzuladığın gibi çalıştığından emin olmanın en iyi yolu budur. Aslında yaptığımız *sizeof()* hakkında dediklerimizi tekrar kontrol etmekti. Örneğin: **double**'a yer ayırmak için basitçe şunu yap:

*double *d = (double *) malloc(sizeof(double));*

Malloc()'un bu çağırımı **sizeof**() için doğru miktarda alan ister; C'de bu genellikle *derleme zamanlı*(**compile-time**)operatör olarak düşünülür, bunun anlamı gerçek boyutu **compile time**'ı bilir ve dolayısıyla bir sayıdır (bu durumda double için 8) **malloc**()'un argumanı olarak değiştirilir. Bu nedenle **sizeof**() doğru bir operatör ve fonskiyon çağırısı değildir(çalışma zamanında bir işlev çağırısı gerçekleşir). Ayrıca **sizeof**() ile bir değişkenin adını iletebilirsiniz (yanlızca türü değil), ancak bazı durumlarda istediğiniz sonuçları alamayabilirsiniz, dikkatli olun. Örneğin, aşağıdaki kod parçacığına bakalım:

*Int *x malloc(10 * sizeof(int));*

Printf("%d/n", sizeof(x));

Ilk satırda, 10 tamsayıdan oluşan bir dizi için boşluk bildirdik, ancak bir sonraki satırda **sizeof**() kullandığımızda, küçük bir değer döndürür, 4 gibi (32 bitlik bir makinada) yada 8 gibi (64 bitlik bir makinada). Bunun nedeni bu durumda **sizeof**() basitçe tam sayının nasıl **pointer**dan büyük

olacağını düşünür, ne kadar hafızamız olduğunu değil. Ancak bazen **sizeof**() tahmin ettiğmiz gibi çalışır

Int x[10];

Printf("%d/n", sizeof(x));

Bu durumda, derleyicinin çalışması için yeterli statik bilgi vardır, 40 baytın ayrıldığını bilin.

Dikkat edilmesi gereken bir diğer yer **string**lerdir. **String** için yer tahsis edildikten sonra, şu programı kullanın: malloc(strlen(s) +1) , buda strlen() işlevini kullanan **string**in sonuna yer açmak için **string**e 1 ekler. **Sizeof**() komutunu kullanmak burda sorun çıkarabilir.

Fark ettiyseniz **malloc**() öğesinde **void** yazmak için **pointer**'ı döndürdüğünüzde. Bunu yapmak C'de bir adresi geri iletmenin ve programcının onunla ne yapmasına karar vermesine izin vermenin bir yoludur.programcı **dökümü** (**cast**) kullanarak daha çok yardımcı olur. Yukarıdaki örneğimizde , programcı **malloc**() dönüş tipini **pointerdan doublea** çevirir. döküm, derleyiciye söylemekten başka bir şey yapmaz ve kodunuzu okuyan diğer programcılar: evet ne yaptığıım biliyorum desede sonuçlar **malloc**() tarafından dökülüyor. Programcı sadece biraz güvence veriyor

Free() komutu

Görünen o ki belleği ayırmak denklemin kolay kısmı; zor olan belleği ne zaman ve nasıl boşaltacağındır. **Heap** bellekte artık kullanılmayan kısmı boşalttığı için programcılar kısaca **free**() der:

*Int *x = malloc(10 * sizeof(int));*

…

Free(x);

program bir arguman alır, **pointer malloc**() tarafından döndürülür. Bu nedenle tahsis edilen bölgenin kullanıcı tarafından içeri aktarılmadığını farkedebilirsiniz ve bellek ayırma kitaplığın kendisi tarafından izlenmelidir.

Yaygın hatalar

Malloc() ve **free**() kullanımında ortaya çıkan yaygın hatalar vardır. Burda tekrar ve terkar gördüğümüz lisans işletim sistemleri dersi. Tüm bu örnekler derleyici tarafından derlenir ve çalıştırılır. Bir C programı derlenirken doğru C programı oluşturmak önemlidir, öğreneceğiniz gibi (genellikle zor yoldan) yeterli olmaktan uzaktır.

Doğru bellek yönetimi öyle bir sorun olmuştur ki, birçok yeni dilde *otomatik bellek yönetim desteği*(**automatic memory management**) vardır. Bu tür dillerde **malloc**() benzeri bir şey çağırırken belleği ayırmak için (genellikle yeni veya yeni bir bellek ayırmaya benzer bir şey) boş alan için çok fazla şey çağırmak zorunda kalmazsınız; daha doğrusu ***garbage collector*** çalışır ve hangi belleğe ihtiyacınız olamadığını bulur ve onu sizin için toplar.

Bellek ayırmayı unutmak

Birçok program siz onları çağırmadan belleğin ayrılmasını bekler. Örneğin *strcpy(dst , src)* programı , hedef **pointer**dan kaynak **pointera** kopyalar. Ancak dikkatli olmazsanız bu yapılabilir:

*char *src = “hello”;*

*char *dts; // opss! Tahsis edilmemiş*

strcpy(dts, src) // hata ve ölüm

Ipucu: derli veya çalışıyor ≠ doğru

Bir programın derlenmiş hatta bir çok kez çalışmış olması doğru olduğu anlamına gelmez. Pek çok olay sizi işe yaradığınız bir noktaya getirmek için bir araya gelmiş olabilir ancak daha sonra bir şeyler değişir ve durur. Ama daha önce işe yaradı yaygın bir öğrenci tepkisidir ve sonrada donanımı, işletim sistemini hatta profesörü suçlayın. Ancak sorun tamda düşündüğünüz yerde yani kodunuzdadır, işe koyul ve başkalarını suçlamadan önce hatalarını ayıkla.

Bu kodu çalıştırdığınızda büyük ihtimalle bir segmentasyon hatasına yol açacaktır, buda HAFIZAYLA BİRLİKTE YANLIŞ BİR ŞEYLER YAPTINIZ SENİ APTAL PROGRAMCI BENİ KIZDIRDIN gibi süslü kelimeler kullanmanıza yol açacaktır

Bu durumda şöyle bir kod uygun olabilir:

*char *src = “hello”;*

*char *dts = (char *) malloc(scrlen(scr) + 1);*

strcpy(dts, src); // düzgün çalışır

Alternatif olarak strdup()’u kullanabilir hayatınızı dahada kolaylaştırabilirsiniz. Daha fazla bilgi için strdup kılavuzunu okuyunuz.

Yeterli bellek ayrılmıyor

Ilgili hata yeterli bellek ayrılamaması, bazen ***buffer overflow***’da denir. Örnekteki yaygın hata hedef buffer’a yeterince boşluk ayrılamamasıdır.

*char *src =”hello”;*

*char *dts = (char *) malloc(scrlen(scr)); //* çok küçük!

strcpy(dts, src); // düzgün çalışır

Işin garibi **malloc**’un nasıl kullanıldığına ve bir çok detaya bağlı olarak, bu program genellikle düzgün çalışır. Bazı durumlarda **string** kopyası yürütüldüğünde bir baytı ayrılmış alanın sonuna yazar, ama bu durumda zararsız, belki kullanılmayan bir değişkenin üzerine yazmak, bazı durumlarda taşmalar aşırı derece zararlı olabilir ve aslında sistemlerdeki bir çok güvenlik açığının kaynağıdır. Başka bir durumda **malloc**() kütüphanesi biraz ekstra alan tahsis ederek başka bir değişkenin değerini çizmez ve gayet iyi çalışır. Öteki durumda program hata verir ve çöker ve böylece yeni bir ders öğrendik: her zaman doğru çalışması demek her zaman doğru değildir.

Ayrılmış hafızayı başlatmayı unutmak

Bu hata ile birlikte düzgünce **malloc**()’u çağırırsınız ancak yeni tahsis edilen verinize değerler girmeyi unutursan. Bunu yapma! Eğer unutursan programınız sonunda başlatılmamış bir okumayla (**uninitialized read**) karşılaşacak. Bu durumda **heap**ten bilinmeyen bazı değerleri okur. Kim bilir

orda ne vardır ? Şanslıysanız programınız hala çalışır (örn. Sıfır) ama değilseniz bir şeyler rastgele ve zararlı.

Belleği boşaltmayı unutmak

Diğer bir yaygın hatada *bellek sızıntısı (**memory leak**)* olarak bilinir, ve belleği boşaltmayı unuttuğunuzda meydana gelir. Uzun süre çalışan programlarda veya sistemlerde (öğreneğin işletim sisteminin kendisi gibi), bu çok büyük bir problemdir. Yavaş yavaş sızan bellek, sonunda kişinin belleğinin tükenmesine neden olur, böyle zamanlarda yeniden başlatma gerekir. Bu nedenle bir **chunk** bellekle işiniz bittiğinde belleği boşalttığınızdan emin olun. Dikkat edin: **garbage-collector**lü bir dil burda yardım edemez: eğer hala **chunk**a sahipseniz , **garbage collector** onu asla boşaltmayacaktır, bu nedenle bellek sızıntıları modern dillerde sorun olmaya devam ediyor. Bazı durumlarda ***free****()* öğesini çağırmamak mantıklı görünebilir, örnek verirsek, programınız kısa ömürlü ve yakında kapanacak; bu durumda, süreç öldüğünde , işletim sistemi kendisine ayrılan tüm sayfaları temizleyecek ve böylece bellek sızıntısı gerçekleşmeyecek. Bu kesinlikle "işe yarasada" muhtemelen geliştirmek için kötü bir alışkanlıktır, bu yüzden böyle bir strateji seçerken dikkatli olun. Uzun vadede, bir programcı olarak amaçlarınızdan birisi iyi alışkanlıklar geliştirmektir; bunlardan biride belleği nasıl yönettiğini ve (C gibi dillerde) blokları boşalttığını ve ayırdığını anlamaktır. Bunu yapmayarak kurtulabilsen bile, sana açıkça tahsis edilmiş her biti nasıl boşaltacağını bilmek iyi bir alışkanlıktır.

Işiniz bitmeden önce belleği boşaltmak

Bazen bir program, kullanımı bitmeden belleği boşaltır; bu hataya ***danglilg pointer(sarkan işaretçi)*** denir. Ve tahmin edeceğiniz gibi bu aynı zamanda kötü bir şey, sonraki kullanımda program çökebilir veya geçerli belleğin üzerine yazar (örneğin: **free**()'yi çağırdınız ama sonra birşeyleri tahsis etmek için tekrar **malloc**()'u çağırdığınız, bu daha sonra hatalı olarak serbest bırakılan belleği geri dönüştürür.)

Belleği defalarca boşaltma

Programlar bazen belleği birden fazla kez boşaltabilir; bu durum ***double free*** olarak bilinir. Bunu yapmanın sonucu tanımsızdır. Tahmin edeceğiniz gibi bellek ayırma kitaplığının kafası karışabilir ve tuhaf şeyler yapabilir; çökmeler yaygın bir sonuçtur.

Yani: işleminiz sonlandıktan sonra neden hiç bellek sızıntısı olmaz

Kısa ömürlü bir program yazdığınızda **malloc**() kullanarak biraz yer ayırabilirsin, program çalışıyor ve tamamlanmak üzere: sonlandırmadan önce sadece birkaç kez **free**()'yi çağırmam lazım mı ? Göründüğü gibi değil, hiçbir hafıza gerçek anlamda kaybolmaz. Sebebi basit: sistemde gerçekte iki bellek yönetim seviyesi vardır. Bellek yönetiminin ilk seviyesi işletim sistemi tarafından gerçekleştirilir, çalıştırıldığında bellek işlemlere dağılır ve işlem bittiğinde geri alınır. Ikinci yönetim seviyesi her işlemin içinde: örneğin **heap**'in içinde **malloc**() ve **free**()'yi çağırdınız. **Free**()'yi çağırmakta başarısız olursanız (ve bu yüzden yığındaki bellek sızıntısı) işletim sistemi tüm belleği geri alacaktır(kod için bu sayfalar dahil, **stack** ve burda ilgili olan **heap**) program çalışmayı durdurduğunda, adres boşluğundaki **heap**'in durumu ne olursa olsun, süreç öldüğünde işletim sistemi tüm bu sayfaları geri alıyor, böylece boşaltılmamış olmasına rağmen hafızanın kaybolmamasını sağlar. Bu nedenle kısa ömürlü programlar için bellek sızıntısı genellikle herhangi bir opsiyonel probleme neden olmaz (zayıf form olarak kabul edilsede). Uzun süre çalışan bir sunucu yazdığınız zaman (asla çıkmayan bir web sunucusu veya veritabanı yönetim sistemi gibi) , bellek sızıntısı çok daha büyük bir sorundur ve uygulama çalışmayı durdurduğunda hafızanın çökmesine

neden olur. Ve tabi ki bellek sızıntısı belirli bir program için daha büyük bir sorundur: işletim sisteminin kendisi. Bize bir kez daha gösteriliyor ki: çekirdek kodunu yazanların işi çok zordur...

Free()'yi yanlış şekilde çağırmak

Tartıştığımız son sorun **free**() öğesinin yanlış çağırılmasıdır. Sonunda **free**() **pointer**lardan birini **malloc**()'tan önce ona iletmemizi ister. Başka bir değere geçtiğinizde kötü şeyler (olur) olabilir. Bu nedenle bu tür ***invalid frees*** *(**geçersiz boşaltma**)* tehlikelidir ve bundan kaçınılmalıdır.

Özet

Gördüğünüz gibi hafızayı kötüye kullanmanın pek çok yolu var. Bellekte ilgili yapılan sık hatalar nedeniyle, kodunuzda bu tür sorunlara yardımcı olacak bütün bir **araç**(**tool**) ekosistemi geliştirilmiştir. **Purify[HJ92]** ve **valgrind[SN05]**'e göz atın, her ikiside hafızayla ilgili problemleri bulmada mükemmeldir. Bu güçlü araçları kullanmaya alıştıktan sonra, onlarsız nasıl hayatta kaldığınızı merak edeceksiniz.

Temel İşletim Sistemi Desteği

malloc() ve **free**() fonksiyonlarını tartışırken sistem çağrılarından bahsetmediğimizi fark etmişsinizdir. Bunun sebebi basittir: onlar sistem çağrıları değil, daha ziyade kütüphane çağrılarıdır. Bu ölçüde **malloc** kütüphanesi sanal adres alanınızdaki alanı yönetir, ancak kendisi daha fazla bellek sormak veya bir kısmını sisteme geri bırakmak için işletim sistemini çağıran bazı sistem çağrılarının üzerine inşa edilmiştir.

Programın *break* konumunu, ***heap*** sonunun konumunu değiştirmek için kullanılan böyle bir sistem çağırısı **brk** olarak adlandırılır: Bir argüman (yeni kesmenin adresi) alır ve böylece yeni kesmenin mevcut kesmeden daha büyük veya daha küçük olmasına bağlı olarak **heap**in boyutunu artırır veya azaltır. Ek olarak **sbrk** çağrısına bir **artış**(**increment**) aktarılır ancak bunun dışında benzer bir amaca hizmet eder.

brk ya da **sbrk**' yi asla doğrudan çağırmamanız gerektiğini unutmayın. Bunlar bellekte yer tahsis etme kütüphaneleri olarak kullanılır; eğer onları kullanmayı denerseniz, muhtemelen bir şeylerin (korkunç bir şekilde) yanlış gitmesine neden olursunuz. Bunun yerine **malloc**() ve **free**() fonksiyonlarına bağlı kalın.

Son olarak, **mmap**() çağrısı aracılığıyla işletim sisteminden de bellek elde edebilirsiniz. Doğru argümanları ileterek, **mmap**() programınız içinde anonim bir bellek bölgesi, belirli bir dosya ile ilişkili olmayan ancak daha sonra sanal bellekte ayrıntılı tartışacağımız **takas alanıyla**(**swap space**) ilişkili bir bölgesi oluşturulabilir. Bu bellek daha sonra bir **heap** olarak alınabilir ve bu şekilde yönetilebilir. Daha fazla ayrıntı için **mmap**()'in kılavuz sayfasını okuyun.

Diğer Çağrılar

Bellek ayırma kütüphanesinin desteklediği birkaç farklı çağrı vardır. Örneğin, **calloc**() bellekte yer ayırır ve aynı zamanda dönüş yapmadan önce sıfırlar; bu belleğin sıfırlandığını varsaydığınız ve kendinizin başlatmayı unuttuğunuz bazı hataları önler (yukarıdaki "başlatılmamış okumalar" paragrafına bakın). **realloc**() rutini, bir şey (örneğin bir dizi) için yer ayırdığınızda ve daha sonra ona

bir şey eklemeye ihtiyacınız olduğunda da kullanışlı olabilir: **realloc**() bellekte daha geniş olan yeni bir bölge oluşturur, eski belleği onun içine kopyalar ve yeni bölgenin **pointerını** döndürür.

Özet

Bellek tahsisi ile ilgili bazı **API**'leri tanıttık. Her zaman olduğu gibi, biz sadece temelleri ele aldık; daha fazla ayrıntı başka yerlerde mevcuttur. Daha fazla bilgi için C kitabını [KR88] ve Stevens [SR05] (Bölüm 7) okuyun. Bu sorunların birçoğunun otomatik olarak nasıl tespit edileceği ve düzeltileceği konusunda güzel ve modern bir makale için Novark ve diğerlerine [N+07] bakın; aynı zamanda bu makale yaygın sorunların güzel bir özetini ve bazı güzel fikirleri de içermektedir.

Ödev (Kod)

Bu ödevde, bellek tahsisi konusunda biraz aşinalık kazanacaksınız. İlk olarak bazı hatalı kodlar yazacaksınız (eğlenceli!). Daha sonra, eklediğiniz hataları bulmanıza yardımcı olacak bazı araçlar kullanacaksınız. Daha sonra, bu araçların ne kadar harika olduğunu fark edecek ve gelecekte onları kullanacaksınız, araçlar hata ayıklayıcıdır (örneğin gdb) ve **valgrind** bir bellek hata bulucudur

Sorular

1. İlk olarak, bir tamsayıya bir işaretçi oluşturan, onu **NULL** olarak ayarlayan ve sonra onu referanssız hale getirmeye çalışan null.c adında basit bir program yazın. Bunu **null** adında bir çalıştırılabilir dosyaya derleyin. Bu programı çalıştırdığınızda ne olur?

Segmentasyon hatası verecek veya çökecektir.

```c
#include<stdio.h>
S
int main(int argc, char **argv) {
    int *x = NULL;
    return 0;
    }
```

2. Ardından, bu programı sembol bilgisini dahil ederek derleyin (-g bayrağı ile). Bunu yapmak, çalıştırılabilir dosyaya daha fazla bilgi koyarak hata ayıklayıcının değişken adları ve benzerleri hakkında daha fazla bilgilere erişmesini sağlar. Programı hata ayıklayıcı altında gdb null yazarak çalıştırın ve gdb çalıştıktan sonra run yazın. gdb size ne gösterir?

```c
#include<stdio.h>

int main(int argc, char **argv) {
    int *x = NULL;
    printf("%d\n" , *x);
    return 0;
    }
```

```
arrowcrown@arrowcrown-VirtualBox:~/null$ gdb null
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from null...
(No debugging symbols found in null)
(gdb) -g
Undefined command: "-g".  Try "help".
(gdb) run
Starting program: /home/arrowcrown/null/null
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555168 in main ()
(gdb)
```

3. Son olarak, bu programda **valgrind** aracını kullanın. **valgrind**'in parçası olan **memcheck** aracını, ne olduğunu analiz etmek için kullanacağız. Aşağıdakileri yazarak çalıştırın: valgrind --leak-check=yes null. Bunu çalıştırdığınızda ne oluyor? Araçtan gelen çıktıyı yorumlayabilir misiniz?

Null.c dosyasının 4. satırında geçersiz okuma olduğunu gösterir.

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

int main(int argc, char **argv) {
    int *x = NULL;
    printf("%d\n" , *x);
    return 0;
    }
```

```
arrowcrown@arrowcrown-VirtualBox:~/null$ valgrind --leak-check=yes ./null
==54046== Memcheck, a memory error detector
==54046== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==54046== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==54046== Command: ./null
==54046==
==54046== Invalid read of size 4
==54046==    at 0x109168: main (in /home/arrowcrown/null/null)
==54046==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==54046==
==54046==
==54046== Process terminating with default action of signal 11 (SIGSEGV)
==54046==  Access not within mapped region at address 0x0
==54046==    at 0x109168: main (in /home/arrowcrown/null/null)
==54046==  If you believe this happened as a result of a stack
==54046==  overflow in your program's main thread (unlikely but
==54046==  possible), you can try to increase the size of the
==54046==  main thread stack using the --main-stacksize= flag.
==54046==  The main thread stack size used in this run was 8388608.
==54046==
==54046== HEAP SUMMARY:
==54046==     in use at exit: 0 bytes in 0 blocks
==54046==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==54046==
==54046== All heap blocks were freed -- no leaks are possible
==54046==
==54046== For lists of detected and suppressed errors, rerun with: -s
==54046== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Parçalama arızası (çekirdek döküldü)
```

4. **malloc**() kullanarak bellek ayıran ancak çıkmadan önce belleği boşaltmayı unutan basit bir program yazın. Bu program çalıştığında ne olur? Herhangi bir sorun bulabilmek için gdb kullanabilir misiniz? Peki ya **valgrind**? (yine –-leak-check=yes bayrağı ile birlikte)?

Gdb bu programdaki sızıntıları tespit etmek için uygun değildir. Bazı hataları bulmamızı sağlayabilir ancak programın belleği boşaltmadığını bize söylemez.

Valgrind ise malloc ile belleğin ayrıldığını ancak daha sonra boşaltılmadığını bizlere gösterir.

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

int main(int argc , char **argvv) {
    int *x = malloc(10 * sizeof(int));
    x[10] = 0;
    return 0;
    }
```

```
arrowcrown@arrowcrown-VirtualBox:~/malloc$ gdb malloc.c
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
"/home/arrowcrown/malloc/malloc.c": not in executable format: file format not r
ecognized
(gdb) run
Starting program:
No executable file specified.
Use the "file" or "exec-file" command.
(gdb)
```

```
arrowcrown@arrowcrown-VirtualBox:~/malloc$ valgrind --leak-check=yes ./malloc
==57502== Memcheck, a memory error detector
==57502== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==57502== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==57502== Command: ./malloc
==57502==
==57502== Invalid write of size 4
==57502==    at 0x109172: main (in /home/arrowcrown/malloc/malloc)
==57502==  Address 0x4a95068 is 0 bytes after a block of size 40 alloc'd
==57502==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-
amd64-linux.so)
==57502==    by 0x109165: main (in /home/arrowcrown/malloc/malloc)
==57502==
==57502==
==57502== HEAP SUMMARY:
==57502==     in use at exit: 40 bytes in 1 blocks
==57502==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==57502==
==57502== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==57502==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-
amd64-linux.so)
==57502==    by 0x109165: main (in /home/arrowcrown/malloc/malloc)
==57502==
==57502== LEAK SUMMARY:
==57502==    definitely lost: 40 bytes in 1 blocks
==57502==    indirectly lost: 0 bytes in 0 blocks
==57502==      possibly lost: 0 bytes in 0 blocks
==57502==    still reachable: 0 bytes in 0 blocks
==57502==         suppressed: 0 bytes in 0 blocks
```

5. **malloc** kullanarak data isminde boyutu 100 olan bir tamsayı dizisi oluşturan bir program yazın; ardından, data[100] değerini sıfıra ayarlayın. Programı çalıştırdığınızda ne olur? Bu programı **valgrind** kullanarak çalıştırdığınızda ne olur? Program doğru mu?

Bu proram çalıştığında bölünememe hatası verecektir veya çökecektir. Valgrind kullanılırsa program ayırmadığı bir kısma erişmeye çalıştığını belirten bir hata mesajı verecektir.

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

int main(int argc , char **argvv){
    int *data;
    data =(int*)malloc(100 * sizeof(int));
    data[100] = 0;
    return 0;
}
```

```
(gdb) exit
arrowcrown@arrowcrown-VirtualBox:~/data$ gdb data
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from data...
(No debugging symbols found in data)
(gdb) run
Starting program: /home/arrowcrown/data/data
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Inferior 1 (process 58966) exited normally]
(gdb)
```

```
arrowcrown@arrowcrown-VirtualBox:~/data$ valgrind --leak-check=yes ./data
==59036== Memcheck, a memory error detector
==59036== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==59036== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==59036== Command: ./data
==59036==
==59036== Invalid write of size 4
==59036==    at 0x109174: main (in /home/arrowcrown/data/data)
==59036==  Address 0x4a951d0 is 0 bytes after a block of size 400 alloc'd
==59036==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-
amd64-linux.so)
==59036==    by 0x109165: main (in /home/arrowcrown/data/data)
==59036==
==59036==
==59036== HEAP SUMMARY:
==59036==
==59036== HEAP SUMMARY:
==59036==     in use at exit: 400 bytes in 1 blocks
==59036==   total heap usage: 1 allocs, 0 frees, 400 bytes allocated
==59036==
==59036== 400 bytes in 1 blocks are definitely lost in loss record 1 of 1
==59036==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-
amd64-linux.so)
==59036==    by 0x109165: main (in /home/arrowcrown/data/data)
==59036==
==59036== LEAK SUMMARY:
==59036==    definitely lost: 400 bytes in 1 blocks
==59036==    indirectly lost: 0 bytes in 0 blocks
==59036==      possibly lost: 0 bytes in 0 blocks
==59036==    still reachable: 0 bytes in 0 blocks
==59036==         suppressed: 0 bytes in 0 blocks
==59036==
==59036== For lists of detected and suppressed errors, rerun with: -s
==59036== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
arrowcrown@arrowcrown-VirtualBox:~/data$
```

6. Bir tamsayılar dizesine yer tahsis eden (yukarıdaki gibi), onları serbest bırakan ve daha sonra dize elemanlarından birinin değerini yazdırmaya çalışan bir program oluşturun. Program çalışıyor mu? Üzerinde **valgrind** kullandığınızda ne olur?

Program sorunsuz bir şekilde çalışır. Valgrind herhangi bir hata vermez.

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

int main(int argc , char **argvv){
   int *data;
   data =(int*)malloc(100 * sizeof(int));
   free(data);
   printf("%d/n" , x[3]);
   return 0;
}
```

```
arrowcrown@arrowcrown-VirtualBox:~/data$ gdb data
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from data...
(No debugging symbols found in data)
(gdb) run
Starting program: /home/arrowcrown/data/data
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Inferior 1 (process 59243) exited normally]
(gdb)
```

```
arrowcrown@arrowcrown-VirtualBox:~/data$ valgrind --leak-check=yes ./data
==59320== Memcheck, a memory error detector
==59320== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==59320== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==59320== Command: ./data
==59320==
==59320== Invalid write of size 4
==59320==    at 0x109174: main (in /home/arrowcrown/data/data)
==59320==  Address 0x4a951d0 is 0 bytes after a block of size 400 alloc'd
==59320==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-
amd64-linux.so)
==59320==    by 0x109165: main (in /home/arrowcrown/data/data)
==59320==
==59320==
```

```
==59320==
==59320== HEAP SUMMARY:
==59320==     in use at exit: 400 bytes in 1 blocks
==59320==   total heap usage: 1 allocs, 0 frees, 400 bytes allocated
==59320==
==59320== 400 bytes in 1 blocks are definitely lost in loss record 1 of 1
==59320==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-
amd64-linux.so)
==59320==    by 0x109165: main (in /home/arrowcrown/data/data)
==59320==
==59320== LEAK SUMMARY:
==59320==    definitely lost: 400 bytes in 1 blocks
==59320==    indirectly lost: 0 bytes in 0 blocks
==59320==      possibly lost: 0 bytes in 0 blocks
==59320==    still reachable: 0 bytes in 0 blocks
==59320==         suppressed: 0 bytes in 0 blocks
==59320==
==59320== For lists of detected and suppressed errors, rerun with: -s
==59320== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
arrowcrown@arrowcrown-VirtualBox:~/data$
```

7. Şimdi **free**'ye komik bir değer atayın (örneğin, yukarıda ayırdığınız dizinin ortasındaki işaretçi). Ne olur? Bu tür sorunları bulmak için araçlara ihtiyacınız var mı?

Malloc atadığımız değerden dolayı farklı bir adresi boşaltmaya çalışıyor.

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

int main(int argc , char **argvv){
   int *data;
   data =(int*)malloc(100 * sizeof(int));
   free(10);
   printf("%d/n" , x[3]);
   return 0;
}
```

```
arrowcrown@arrowcrown-VirtualBox:~/data$ gcc -g data.c
data.c: In function ‘main’:
data.c:7:9: warning: passing argument 1 of ‘free’ makes pointer from integer wi
thout a cast [-Wint-conversion]
    7 |     free(10);
      |          ^~
      |          |
      |          int
In file included from data.c:2:
/usr/include/stdlib.h:555:25: note: expected ‘void *’ but argument is of type ‘
int’
  555 | extern void free (void *__ptr) __THROW;
      |                   ~~~~~~~^~~~~
data.c:8:20: error: ‘x’ undeclared (first use in this function)
    8 |     printf("%d/n" , x[3]);
      |                      ^
data.c:8:20: note: each undeclared identifier is reported only once for each fu
nction it appears in
arrowcrown@arrowcrown-VirtualBox:~/data$
```

8. Bellek tahsisine yönelik diğer arayüzlerden bazılarını deneyin. Örneğin, basit bir vektör benzeri veri yapısı ve vektörü yönetmek için **realloc**() kullanan ilgili rutinler oluşturun. Vektör elemanlarını saklamak için bir dizi kullanın; kullanıcı vektöre bir giriş eklediğinde, daha fazla yer tahsis etmek için **realloc**() fonksiyonunu kullanın. Böyle bir vektör ne kadar iyi performans gösterir? Bağlı liste ile karşılaştırıldığında nasıldır? Hataları bulmada yardımcı olması için **valgrind** kullanın.

Küçük vektörler için realloc() kullanmak iyi bir performans gösterebilir ancak vektör büyüdükçe diziyi yeniden boyutlamdırmak zahmetli ve maliyetli bir hale gelecektir. Bağlı liste iste büyük vektörlerde daya iyi bir performans gösterebilir.

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

int main(void)
{
 int *ip, id;
 ip = (int*) malloc(5 * sizeof(int));

 for (id=0; id<5; id++){
      *(ip+id) = id+1;
      printf("%p adresindeki değer: %d/n", (ip+id), *(ip+id));
      }
 ip = (int*) realloc(ip,10 * sizeof(int));
 printf("genişletilmiş bellek değerleri:/n");
 for ( ; id<10; id++) {
 *(ip+id) = id+1;
 printf("%p adresindeki değer: %d/n" , (ip+id), *(ip+id));
 }
 free(ip);
 return 0;
 }

```

9. **gdb** ve **valgrind**'i kullanmak hakkında okumaya daha fazla zaman harcayın. Araçlarınızı bilmek kritiktir; **UNIX**'de ve C ortamında nasıl uzman bir hata ayıklayıcı olunur hakkında zaman harcayın ve öğrenin.