14

Interlude: Bellek API'sı

Bu aralıkta, UNIX sistemlerindeki bellek ayırma arayüzlerini tartışacağız . Sağlananarayüzler oldukça basittir ve bu nedenle bölüm kısa ve nokta atışı . Ele aldığımız temel sorun şudur:

CRUX : BELLEK NASIL AYIRILIR VE YÖNETİLİR

UNIX/C programlarında , belleğin nasıl tahsis edileceğini ve yönetileceğini anlamak, sağlam ve güvenilir yazılım oluşturmak için çok önemlidir. Yaygın olarak hangi ara yüzler kullanılır? Hangi hatalardan kaçınılmalıdır?

14.1 Bellek Türleri

Bir C programını çalıştırırken, ayrılan iki tür bellek vardır. İlki , **yığın(stack)** belleği olarakadlandırılır ve bunun tahsisleri ve serbest tahsisleri, programcı olarak sizin içinderleyici tarafından dolaylı olarak yönetilir; bu nedenle bazen **otomatik** bellekolarak adlandırılır .

C'de yığındaki belleği bildirmek kolaydır. Örneğin, x adlı bir tamsayı için *func()* işlevinde biraz boşluğa ihtiyacınız olduğunu varsayalım. Böyle bir anıyı ilan etmek için şöyle bir şey yapmanız yeterlidir:

```
void func() {
        int x; // yığında bir tamsayı bildirir
    ...
}
```

Derleyici, func() işlevini çağırdığınızda yığında yer açtığınızdan emin olarak gerisini halleder. İşlevden döndüğünüzde, derleyici belleği sizin için yeniden tahsis eder; bu nedenle, bazı bilgilerin çağrı çağrısının ötesinde yaşamasını istiyorsanız, bu bilgiyi yığında bırakmamanız daha iyi olur

Uzun ömürlü belleğe duyulan bu ihtiyaç, bizi , tüm ayırmaların ve ayırmaların açıkça siz programcı tarafından işlendiği **yığın(heap)** bellek adı verilen ikinci tür belleğe götürür . Ağır bir sorumluluk şüphesiz! Ve kesinlikle birçok hatanın nedeni. Ama dikkatli olursanız ve dikkat ederseniz, bu tür arayüzleri doğru ve çok fazla sorun yaşamadan kullanırsınız. Öbek üzerinde bir tam sayının nasıl tahsis edilebileceğine dair bir örnek:

```
void func() {
    int *x = (int *) malloc(sizeof(int));
    ...
}
```

Bu küçük kod parçacığı hakkında birkaç not. İlk olarak, bu satırda hem yığın hem de yığın tahsisinin gerçekleştiğini fark etmeyebilirsiniz: ilk olarak derleyici, söz konusu işaretçiye (int *x) ilişkin bildiriminizi gördüğünde bir tamsayıya işaretçiye yer açmasını bilir; daha sonra, program malloc()'u çağırdığında öbek üzerinde bir tamsayı için boşluk ister; rutin böyle bir tamsayının adresini döndürür (başarı durumunda veya başarısızlık durumunda NULL), bu daha sonra program tarafından kullanılmak üzere yığında depolanır.

Açık doğası ve daha çeşitli kullanımı nedeniyle yığın bellek, hem kullanıcılar hem de sistemler için daha fazla zorluk sunar. Bu nedenle, tartışmamızın geri kalanının odak noktası budur.

14.2 malloc() Çağrısı

malloc() çağrısı oldukça basittir: onu öbek üzerinde bir yer isteyen bir boyut iletirsinizve bu ya başarılı olur ve size yeni ayrılan alana bir işaretçi verir ya da başarısız olur ve NULL2 değerini döndürür .

Kılavuz sayfası, malloc'u kullanmak için yapmanız gerekenleri gösterir; tip adam komut satırında malloc ve şunu göreceksiniz:

```
#include <stdlib.h>
...
void *malloc(size_t boyut);
```

Bu bilgilerden, malloc'u kullanmak için yapmanız gereken tek şeyin stdlib.h başlık dosyasını dahil etmek olduğunu görebilirsiniz. Aslında, tüm C programlarının varsayılan olarak bağlandığı C kitaplığının içinde malloc() kodu bulunduğundan, bunu gerçekten yapmanıza bile gerek yoktur; başlığın eklenmesi derleyicinin malloc() işlevini doğru çağırıp çağırmadığınızı kontrol etmesini sağlar (örn. ona doğru sayıda, doğru türden argüman iletmek).

malloc()'un aldığı tek parametre boyutu t olup, kaç bayta ihtiyacınız olduğunu basitçe tanımlar. Ancak çoğu programcı buraya doğrudan bir sayı yazmaz (10 gibi); aslında, bunu yapmak kötü bir biçim olarak kabul edilir. Bunun yerine, çeşitli rutinler ve makrolar

2C'deki NULL'un gerçekten özel bir şey olmadığını, yalnızca sıfır değeri için bir makro olduğunu unutmayın.

İPUCU: ŞÜPHE DURUMUNDA DENEYİN

Kullandığınız bir rutinin veya operatörün nasıl davrandığından emin değilseniz, onudeneyip beklediğiniz gibi davrandığından emin olmanın yerini hiçbir şey tutamaz . Kılavuz sayfalarını veya diğer belgeleri okumak yararlı olsa da, pratikte nasıl çalıştığıönemlidir. Biraz kod yazın ve test edin! Kodunuzun istediğiniz gibi davrandığından emin olmanın en iyi yolu şüphesiz budur. Aslında, sizeof() hakkında söylediğimiz şeylerin gerçekten doğru olup olmadığını tekrar kontrol etmek için yaptığımız şey buydu!

kullanıldı. Örneğin, çift duyarlıklı bir kayan nokta değeri için alan ayırmak için, basitçe şunu yapın:

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

Vay canına, bu çok fazla ikileme! Bu malloc() çağrısı, doğru miktarda alan istemek için sizeof() operatörünü kullanır; C'de bu genellikle bir derleme zamanı işleci olarak düşünülür, yani gerçek boyutun derleme zamanında bilindiği ve bu nedenle malloc()'un argümanı olarak bir sayının (bu durumda bir çift için 8) ikame edildiği anlamına gelir. Bu nedenle, sizeof() doğru bir şekilde bir işlev çağrısı değil bir operatör olarak düşünülür (çalışma zamanında bir işlev çağrısı gerçekleşir).

Sizeof() işlevine bir değişkenin adını da (yalnızca bir türü değil) iletebilirsiniz, ancak bazı durumlarda istenen sonuçları alamayabilirsiniz, bu yüzden dikkatli olun. Örneğin, aşağıdaki kod parçacığına bakalım:

```
int *x = malloc(10 * sizeof(int));
printf("%d\n",sizeof(x));
```

İlk satırda, 10 tam sayıdan oluşan bir dizi için boşluk bildirdik ki bu iyi ve şık. Ancak bir sonraki satırda sizeof() kullandığımızda 4 (32 bit makinelerde) veya 8 (64 bit makinelerde) gibi küçük bir değer döndürür. Bunun nedeni, bu durumda sizeof()'un dinamik olarak ne kadar bellek ayırdığımızı değil, bir tamsayıya işaretçinin ne kadar büyük olduğunu sorduğumuzu düşünmesidir. Ancak bazen sizeof() beklediğiniz gibi çalışır:

```
int x[10];
printf("%d\n", sizeof(x));
```

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

Dikkat edilmesi gereken bir diğer yer de iplerdir. Bir dizge için boşluk bildirirken, şu deyimi kullanın: malloc(strlen(s) + 1), bu deyim strlen() işlevini kullanarak dizenin uzunluğunu alır ve sona yer açmak için diziye 1 ekler. dize karakteri. sizeof() işlevinin kullanılması burada sorunlara yol açabilir.

Malloc() öğesinin void yazmak için bir işaretçi döndürdüğünü de fark edebilirsiniz. Bunu yapmak, C'de bir adresi geri göndermenin ve onunla ne yapacağına programcının karar vermesine izin vermenin yoludur. Programcı ayrıca, oyuncu kadrosu denen şeyi kullanarak yardımcı olur ; Yukarıdaki örneğimizde, programcı malloc() dönüş tipini bir çift işaretçiye çevirir. Döküm, derleyiciye ve kodunuzu okuyor olabilecek diğer programcılara "evet, ne yaptığımı biliyorum" demek dışında gerçekten hiçbir şey başaramaz. malloc() sonucunu yayınlayarak, programcı sadece biraz güvence veriyor; doğruluk için alçıya gerek yoktur.

14.3 free()

Görünen o ki, bellek ayırmak denklemin kolay kısmı; ne zaman, nasıl ve hatta belleği boşaltıp boşaltmayacağını bilmek zor kısımdır. Artık kullanılmayan yığın belleği boşaltmak için programcılar basitçe **free()** öğesini çağırır:

```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

Rutin, malloc() tarafından döndürülen bir işaretçi olan bir bağımsız değişken alır. Bu nedenle, ayrılan bölgenin boyutunun kullanıcı tarafından aktarılmadığını ve bellek ayırma kitaplığının kendisi tarafından izlenmesi gerektiğini fark edebilirsiniz.

14.4 Yaygın Hatalar

malloc() ve free() kullanımında ortaya çıkan bir dizi yaygın hata vardır. Lisans işletim sistemleri dersini verirken defalarca gördüğümüz bazılarını burada bulabilirsiniz. Tüm bu örnekler, derleyiciden hiç ses çıkarmadan derlenir ve çalıştırılır; doğru bir C programı oluşturmak için bir C programı derlemek gerekli olsa da, öğreneceğiniz gibi (genellikle zor yoldan) yeterli olmaktan uzaktır.

Doğru bellek yönetimi öyle bir sorun olmuştur ki, aslında birçok yeni dil **otomatik bellek yönetimini** desteklemektedir . Bu tür dillerde, bellek ayırmak için malloc() benzeri bir şey çağırırken (genellikle yeni veya yeni bir nesne ayırmak için benzer bir şey), boş alan için hiçbir zaman bir şey çağırmanız gerekmez; bunun yerine, bir çöp toplayıcı çalışır ve artık hangi belleğe başvurmadığınızı bulur ve onu sizin için boşaltır.

Belleği Tahsis Etmeyi Unutmak

Bir çok rutin, siz onları çağırmadan önce belleğin ayrılmasını bekler. Örneğin, strcpy(dst, src) yordamı, kaynak işaretçiden hedef işaretçiye bir dize kopyalar. Ancak dikkatli olmazsanız şunu yapabilirsiniz:

```
 char *src = "hello";

Char *dst;   // oops! ayrılmamış hafıza

strcpy(dst, src); // segfault ve öl
```

İPUCU: DERLEDİ VEYA ÇALIŞTI = DOĞRU Bir programın derlenmesi(!), hatta bir veya birçok kez doğru çalıştırılması, programın doğru olduğu anlamına gelmez . Pek çok olay, sizi işe yaradığına inandığınız bir noktaya getirmek için işbirliği yapmış olabilir, ancak sonra bir şeyler değişir ve durur. Yaygın bir öğrenci tepkisi söylemek (veya bağırmaktır)
"Ama daha önce işe yaradı!" ve sonra derleyiciyi, işletim sistemini, donanımı ve hatta (söylemeye cüret edelim) profesörü suçlayın. Ancak sorun genellikle tam da olacağını düşündüğünüz yerde, kodunuzdadır. Diğer bileşenleri suçlamadan önce çalışmaya başlayın ve hatalarını ayıklayın.

Bu kodu çalıştırdığınızda, muhtemelen **BELLEK İLE BİR ŞEYİ YANLIŞ YAPTINIZ , APTAL PROGRAMCI VE BEN KIZGINIM** için süslü bir terim olan birsegmentasyon hatasına yol açacaktır.

Bu durumda, uygun kod şöyle görünebilir:

```
char *src = "hello";

char *dst = (char *) malloc(strlen(src) + 1);

strcpy(dst, kaynak); // düzgünçalışır
```

Alternatif olarak, strdup()'u kullanabilir ve hayatınızı daha da kolaylaştırabilirsiniz. Daha fazla bilgi için strdup kılavuz sayfasını okuyun.

Yeterli Bellek Ayrılmıyor

İlgili bir hata,bazen **arabellek taşması(buffer overflow)** olarak adlandırılan yeterli bellek ayrılmamasıdır .
Yukarıdaki örnekte, yaygın bir hata, hedef tampon için neredeyse yeterli yer açmaktır.

```
char *src = "hello";
char*dst = (char *) malloc(strlen(src)); // çok küçük!
strcpy(dst, kaynak); // düzgün çalışır
```

İşin garibi, malloc'un nasıl uygulandığına ve diğer birçok ayrıntıya bağlı olarak, bu program genellikle görünüşte doğru şekilde çalışacaktır. Bazı durumlarda, dize kopyası yürütüldüğünde, ayrılan alanın sonundan bir bayt çok uzağa yazar, ancak bazı durumlarda bu zararsızdır, belki de artık kullanılmayan bir değişkenin üzerine yazılır. Bazı durumlarda, bu aşırı akışlar inanılmaz derecede zararlı olabilir ve aslında sistemlerdeki birçok güvenlik açığının kaynağıdır [W06]. Diğer durumlarda, malloc kitaplığı zaten biraz fazladan alan ayırdı ve bu nedenle programınız aslında başka bir değişkenin değerini karalamıyor ve oldukça iyi çalışıyor. Diğer durumlarda bile, program gerçekten hata verecek ve çökecektir. Ve böylece başka bir değerli ders öğreniyoruz: Bir kez doğru çalışması, onun doğru olduğu anlamına gelmez.

3Gizemli görünse de, bu tür yasa dışı bir bellek erişiminin neden yanlış olduğunu yakında öğreneceksiniz. segmentasyon hatası olarak adlandırılır; Bu okumaya devam etmek için teşvik değilse, nedir?

Tahsis Edilen Belleği Başlatmayı Unutmak

Bu hatayla,malloc()'u düzgün bir şekilde çağırırsınız, ancak yeni tahsis edilen veri tipinize bazı değerleri girmeyi unutursunuz. Bunu yapma! Unutursanız, programınız eninde sonunda yığından değeri bilinmeyen bazı verileri okuduğu **başlatılmamış bir okumayla(uninitialized read)** karşılaşır. Orada ne olabileceğini kim bilebilir? Şanslıysanız, programın hala çalışacağı bir değer (örneğin, sıfır). Şanslı değilseniz, rastgele ve zararlı bir şey.

Belleği Boşaltmayı Unutma Diğer

bir yaygın hata **bellek sızıntısı(memory leak)** olarak bilinir ve belleği boşaltmayı unuttuğunuzda ortaya çıkar. Uzun süre çalışan uygulamalarda veya sistemlerde (işletim sisteminin kendisi gibi), bu çok büyük bir sorundur, çünkü yavaş yavaş sızan bellek, hatta
sonunda kişinin belleğinin bitmesine neden olur ve bu noktada yeniden başlatma gerekir. Bu nedenle, genel olarak, bir yığın bellekle işiniz bittiğinde, onu boşalttığınızdan emin olmalısınız. Çöpten toplanan bir dil kullanmanın burada yardımcı olmadığına dikkat edin: Hala bir miktar bellek yığınına referansınız varsa, hiçbir çöp toplayıcı onu serbest bırakamaz ve bu nedenle bellek sızıntıları daha modern dillerde bile bir sorun olmaya devam eder.

Bazı durumlarda, free() öğesini çağırmamak makul görünebilir. Örneğin, programınız kısa ömürlü ve yakında kapanacak; bu durumda, işlem sona erdiğinde, işletim sistemi kendisine tahsis edilen tüm sayfaları temizleyecek ve böylece kendi başına bellek sızıntısı oluşmayacaktır. Bu kesinlikle "işe yarasa da" (7. sayfadaki kenara bakın), muhtemelen geliştirilmesi kötü bir alışkanlıktır, bu nedenle böyle bir strateji seçerken dikkatli olun. Uzun vadede, bir programcı olarak hedeflerinizden biri iyi alışkanlıklar geliştirmektir; bu alışkanlıklardan biri, belleği nasıl yönettiğinizi anlamak ve (C gibi dillerde) ayırdığınız blokları serbest bırakmaktır. Bunu yapmamaktan kurtulabilseniz bile, açıkça ayırdığınız her baytı serbest bırakma alışkanlığını edinmek muhtemelen iyidir.

İşlemi Bitirmeden Belleği Boşaltmak

Bazen bir program,kullanımı bitmeden önce belleği boşaltır; böyle bir hataya **sarkan işaretçi(dangling pointer)** denir ve tahmin edebileceğiniz gibi bu da kötü bir şeydir. Sonraki kullanım, programı çökertebilir veya geçerli belleğin üzerine yazabilir (örneğin, free() öğesini çağırdınız,ancak daha sonra başka bir şey ayırmak için malloc() öğesini yeniden çağırdınız ve bu da hatalı olarak serbest bırakılan belleği geri dönüştürür).

Belleği Defalarca Boşaltma

Programlarayrıca bazen belleği birden çok kez boşaltır; bu **çift serbest(double free)** olarak bilinir . Bunu yapmanın sonucu tanımsızdır. Tahmin edebileceğiniz gibi, bellek ayırma kitaplığınınkafası karışabilir ve bir sürü tuhaf şey yapabilir; çökmeler yaygın bir sonuçtur.

YANİ : İŞLEMİNİZ ÇIKTIKTAN SONRA NEDEN HİÇ BELLEK SIZMAZ _

Kısa ömürlü bir program yazdığınızda malloc() kullanarak biraz yer ayırabilirsiniz. Program çalışıyor ve tamamlanmak üzere: Çıkmadan hemen önce free() öğesini birkaç kez çağırmanız gerekiyor mu? Bunu yapmamak yanlış gibi görünse de, hiçbir anı gerçek anlamda "kaybolmaz". Nedeni basit: sistemde gerçekten iki bellek yönetimi düzeyi vardır.

Bellek yönetiminin ilk düzeyi, işletim sistemi tarafından gerçekleştirilir; bu, çalıştıklarında işlemlere bellek dağıtır ve işlemler bittiğinde (veya başka bir şekilde öldüğünde) geri alır. İkinci yönetim seviyesi her sürecin içindedir, örneğin malloc() ve free()'yi çağırdığınızda yığının içindedir. free() işlevini çağıramasanız bile (ve dolayısıyla öbekte bellek sızdırıyor olsanız bile), program program açıldığında işletim sistemi işlemin tüm belleğini (kod, yığın ve burada ilgili olduğu şekilde yığın için olan sayfalar dahil) geri alacaktır. çalıştırılarak bitirilir. Adres alanınızdaki yığınınızın durumu ne olursa olsun, işlem sona erdiğinde işletim sistemi tüm bu sayfaları geri alır, böylece belleği boşaltmamış olmanıza rağmen hiçbir belleğin kaybolmamasını sağlar.

Bu nedenle, kısa ömürlü programlar için, bellek sızıntısı genellikle herhangi bir işletim sorununa neden olmaz (ancak kötü biçim olarak kabul edilebilir). Uzun süre çalışan bir sunucu (hiç çıkmayan bir web sunucusu veya veritabanı yönetim sistemi gibi) yazdığınızda, sızan bellek çok daha büyük bir sorundur ve uygulamanın belleği bittiğinde eninde sonunda çökmeye neden olur. Ve elbette, bellek sızıntısı belirli bir programda daha da büyük bir sorundur: işletim sisteminin kendisi. Bize bir kez daha gösteriyoruz: çekirdek kodunu yazanların işi en zor olanlardır...

free()'yi Yanlış Olarak Çağırmak

Tartıştığımız son bir problem de free()'nin yanlış çağrılması. Ne de olsa, free() sizden yalnızca daha önce malloc()'tan aldığınız işaretçilerden birini ona iletmenizi bekler. Başka bir değer aktardığınızda, kötü şeyler olabilir (ve olur). Bu nedenle, bu tür **geçersiz serbest bırakmalar(invalid frees)** tehlikelidir ve elbette bundanda kaçınılmalıdır.

Özet

Gördüğünüz gibi, hafızayı kötüye kullanmanın pek çok yolu var. Bellekle ilgili sık sık yapılan hatalar nedeniyle, kodunuzdaki bu tür sorunları bulmanıza yardımcı olacak koca bir araç ekosferi gelişmiştir. Hem purify [HJ92] hem de valgrind [SN05]' e bakın ; her ikisi de bellekle ilgili sorunlarınızın kaynağını bulmanıza yardımcı olmakta mükemmeldir. Bu güçlü araçları kullanmaya alıştığınızda, onlarsız nasıl hayatta kaldığınızı merak edeceksiniz.

14.5 Temel İşletim Sistemi Desteği

malloc() ve free()'den bahsederken sistem çağrılarından bahsetmediğimizi fark etmişsinizdir. Bunun nedeni basit: Bunlar sistem çağrıları değil, kütüphane çağrılarıdır. Böylece malloc kitaplığı, sanal adres alanınızdaki alanı yönetir, ancak kendisi, daha fazla bellek istemek veya bir kısmını sisteme geri bırakmak için işletim sistemini çağıran bazı sistem çağrılarının üzerine inşa edilmiştir.

Böyle bir sistem çağrısına brk adı verilir ve program kesintisinin konumunu, yani yığının sonunun konumunu değiştirmek için kullanılır . Bir bağımsız değişken (yeni kesmenin adresi) alır ve böylece yeni kesmenin mevcut kesmeden daha büyük veya daha küçük olmasına bağlı olarak yığının boyutunu artırır veya azaltır. Ek bir çağrı sbrk bir artış iletilir, ancak bunun dışında benzer bir amaca hizmet eder.

Asla brk veya sbrk'yi doğrudan aramamanız gerektiğini unutmayın. Bellek ayırma kitaplığı tarafından kullanılırlar; onları kullanmaya çalışırsanız, muhtemelen bir şeylerin (korkunç) ters gitmesine neden olursunuz. Bunun yerine malloc() ve free()'ye bağlı kalın.

Son olarak, mmap() çağrısı yoluyla işletim sisteminden de bellek alabilirsiniz. Doğru bağımsız değişkenleri ileterek, mmap() programınızda anonim bir bellek bölgesi yaratabilir - herhangi bir dosyayla değil, takas alanıyla ilişkili bir bölge, daha sonra sanal bellekte ayrıntılı olarak tartışacağımız bir şey. Bu bellek daha sonra bir yığın gibi ele alınabilir ve bu şekilde yönetilebilir. Daha fazla ayrıntı için mmap() kılavuz sayfasını okuyun.

14.6 Diğer Aramalar

Bellek ayırma kitaplığının desteklediği birkaç başka çağrı vardır. Örneğin, calloc() belleği ayırır ve geri döndürmeden önce onu sıfırlar; bu, belleğin sıfırlandığını varsaydığınız ve belleği kendiniz başlatmayı unuttuğunuz bazı hataları önler (yukarıdaki "başlatılmamış okumalar" ile ilgili paragrafa bakın). Rutin realloc(), bir şey için (mesela bir dizi) yer ayırdığınızda ve sonra buna bir şey eklemeniz gerektiğinde de yararlı olabilir: realloc() bellekte daha büyük yeni bir bölge oluşturur, eski bölgeyi içine kopyalar. ve işaretçiyi yeni bölgeye döndürür.

14.7 Özet

Bellek tahsisi ile ilgili bazı API'leri tanıttık.
Her zaman olduğu gibi, sadece temelleri ele aldık; daha fazla detay başka bir yerde mevcuttur. Daha fazla bilgi için C kitabı [KR88] ve Stevens [SR05] (Bölüm 7) okuyun. Bu sorunların birçoğunun otomatik olarak nasıl tespit edilip düzeltileceğine dair havalı ve modern bir makale için bkz. Noark ve ark. [N+07]; Bu makale ayrıca yaygın sorunların güzel bir özetini ve bunların nasıl bulunup düzeltileceğine dair bazı güzel fikirler içermektedir.

Referanslar

[HJ92] "Purify: Fast Detection of Memory Leaks and Access Errors" by R. Hastings, B. Joyce. USENIX Winter '92. *The paper behind the cool Purify tool, now a commercial product.*

[KR88] "The C Programming Language" by Brian Kernighan, Dennis Ritchie. Prentice-Hall 1988. *The C book, by the developers of C. Read it once, do some programming, then read it again, and then keep it near your desk or wherever you program.*

[N+07] "Exterminator: Automatically Correcting Memory Errors with High Probability" by G. Novark, E. D. Berger, B. G. Zorn. PLDI 2007, San Diego, California. *A cool paper on finding and correcting memory errors automatically, and a great overview of many common errors in C and C++ programs. An extended version of this paper is available CACM (Volume 51, Issue 12, December 2008).*

[SN05] "Using Valgrind to Detect Undefined Value Errors with Bit-precision" by J. Seward, N. Nethercote. USENIX '05. *How to use valgrind to find certain types of errors.*

[SR05] "Advanced Programming in the UNIX Environment" by W. Richard Stevens, Stephen
A. Rago. Addison-Wesley, 2005. *We've said it before, we'll say it again: read this book many times and use it as a reference whenever you are in doubt. The authors are always surprised at how each time they read something in this book, they learn something new, even after many years of C programming.*

[W06] "Survey on Buffer Overflow Attacks and Countermeasures" by T. Werthman. Avail- able: www.nds.rub.de/lehre/seminar/SS06/Werthmann BufferOverflow.pdf. *A nice survey of buffer overflows and some of the security problems they cause. Refers to many of the famous exploits.*

Ödev (Kod)

Bu ev ödevinde, bellek ayırma konusunda biraz aşinalık kazanacaksınız. İlk olarak, bazı hatalı programlar yazacaksınız (eğlenceli!). Ardından, eklediğiniz hataları bulmanıza yardımcı olacak bazı araçlar kullanacaksınız. O zaman bu araçların ne kadar harika olduğunu fark edecek ve gelecekte onları kullanacak, böylece kendinizi daha mutlu ve üretken kılacaksınız. Araçlar, hata ayıklayıcı (örneğin, gdb) ve valgrind [SN05] adı verilen bir bellek hatası algılayıcıdır.

Sorular

1. Önce, bir tamsayıya işaretçi oluşturan, onu NULL'a ayarlayan ve sonra onun başvurusunu kaldırmaya çalışan null.c adında basit bir program yazın. Com, bunu null adlı bir yürütülebilir dosyaya yığın. Bu programı çalıştırdığınızda ne olur?

2. Ardından, bu programı dahil edilen sembol bilgileriyle (-g bayrağıyla) derleyin. Bunu yaparak, hata ayıklayıcının değişken adları ve benzerleri hakkında daha yararlı bilgilere erişmesini sağlayarak yürütülebilir dosyaya daha fazla bilgi koyalım. gdb null yazarak ve gdb çalıştıktan sonra run yazarak programı hata düzeltici altında çalıştırın. gdb size ne gösteriyor?

3. Son olarak bu programda valgrind aracını kullanın. Ne olduğunu analiz etmek için valgrind'in bir parçası olan memcheck aracını kullanacağız. Bunu aşağıdakini yazarak çalıştırın: valgrind --leak-check=yes null. Bunu çalıştırdığınızda ne olur? Aracın çıktısını yorumlayabilir misiniz?

4. Malloc() kullanarak belleği ayıran ancak çıkmadan önce belleği boşaltmayı unutan basit bir program yazın. Bu program çalıştığında ne olur? Bununla ilgili herhangi bir sorun bulmak için gdb'yi kullanabilir misiniz? Valgrind'e ne dersiniz (yine --leak-check=yes bayrağıyla)?

5. malloc kullanarak 100 boyutunda veri adı verilen bir tamsayı dizisi oluşturan bir program yazın; ardından data[100]'ü sıfır olarak ayarlayın. Bu programı çalıştırdığınızda ne olur? Bu programı valgrind kullanarak çalıştırdığınızda ne olur? Program doğru mu?

6. Bir tamsayı dizisi tahsis eden (yukarıdaki gibi), onları serbest bırakan ve ardından dizinin öğelerinden birinin değerini yazdırmaya çalışan bir program oluşturun. Program çalışıyor mu? Üzerinde valgrind kullandığınızda ne olur?

7. Şimdi serbest olarak komik bir değer iletin (örneğin, yukarıda ayırdığınız dizinin ortasındaki bir işaretçi). Ne oluyor? Bu tür bir sorunu bulmak için araçlara ihtiyacınız var mı?

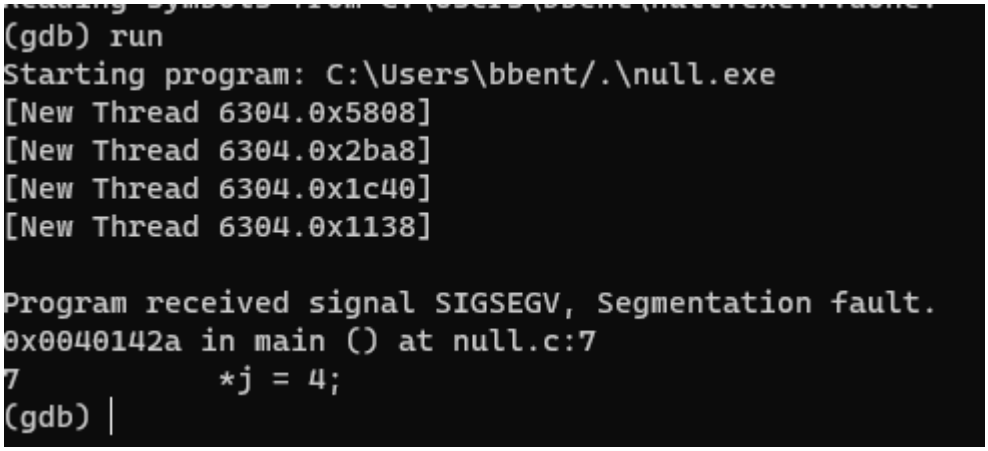
8. Bellek tahsisi için diğer bazı arabirimleri deneyin. Örneğin, vektörü yönetmek için realloc() kullanan basit bir vektör benzeri veri yapısı ve ilgili rutinler oluşturun. Vektör öğelerini depolamak için bir dizi kullanın; Bir kullanıcı vektöre bir giriş eklediğinde, buna daha fazla yer ayırmak için realloc()'u kullanın. Böyle bir vektör ne kadar iyi performans gösterir? Bağlantılı bir listeyle nasıl karşılaştırılır? Hataları bulmanıza yardımcı olması için valgrind kullanın.

9. Daha fazla zaman ayırın ve gdb ve valgrind'i kullanma hakkında bilgi edinin. Aletlerinizi bilmek çok önemlidir; vakit geçirin ve UNIX ve C ortamında nasıl uzman bir hata ayıklayıcı olunacağını öğrenin .

1.Önce, bir tamsayıya işaretçi oluşturan, onu NULL'a ayarlayan ve sonra onun başvurusunu kaldırmaya çalışan null.c adında basit bir program yazın. Bunu null adlı bir yürütülebilir dosyaya derleyin. Bu programıçalıştırdığınızda ne olur?

Çıktı vermiyor

2.Ardından, bu programı dahil edilen sembol bilgileriyle (-g bayrağıyla) derleyin. Bunu yaparak, hata ayıklayıcının değişken adları ve benzerleri hakkında daha yararlı bilgilere erişmesini sağlayarak yürütülebilir dosyayadaha fazla bilgi koyalım. gdb null yazarak ve gdb çalıştıktan sonra run yazarak programı hata düzeltici altında çalıştırın. gdb size ne gösteriyor?



Segmentation Fault

3.Son olarak bu programda valgrind aracını kullanın. Ne olduğunu analiz etmek için valgrind'in bir parçası olan memcheck aracını kullanacağız. Bunu aşağıdakini yazarak çalıştırın: valgrind -leak-check=yes null. Bunu çalıştırdığınızda ne olur? Aracın çıktısını yorumlayabilir misiniz?

```
==903== Memcheck, a memory error detector
==903== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==903== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==903== Command: ./null
==903==
==903== Invalid write of size 4
==903==    at 0x109161: main (null.c:5)
==903==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==903==
==903==
==903== Process terminating with default action of signal 11 (SIGSEGV)
==903==  Access not within mapped region at address 0x0
==903==    at 0x109161: main (null.c:5)
==903==  If you believe this happened as a result of a stack
==903==  overflow in your program's main thread (unlikely but
==903==  possible), you can try to increase the size of the
==903==  main thread stack using the --main-stacksize= flag.
==903==  The main thread stack size used in this run was 8388608.
==903==
==903== HEAP SUMMARY:
==903==     in use at exit: 0 bytes in 0 blocks
==903==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==903==
==903== All heap blocks were freed -- no leaks are possible
==903==
==903== For lists of detected and suppressed errors, rerun with: -s
==903== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault
```

Bellek sızıntımız yok

4.Malloc() kullanarak belleği ayıran ancak çıkmadan önce belleği boşaltmayı unutan basit bir program yazın. Bu program çalıştığında ne olur? Bununlailgili herhangi bir sorun bulmak için gdb'yi kullanabilir misiniz? Valgrind'e ne dersiniz (yine --leak-check=yes bayrağıyla)?

```
==962== Memcheck, a memory error detector
==962== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==962== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==962== Command: ./malloc
==962==
==962==
==962== HEAP SUMMARY:
==962==     in use at exit: 400 bytes in 1 blocks
==962==   total heap usage: 1 allocs, 0 frees, 400 bytes allocated
==962==
==962== 400 bytes in 1 blocks are definitely lost in loss record 1 of 1
==962==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==962==    by 0x10915E: main (malloc.c:4)
==962==
==962== LEAK SUMMARY:
==962==    definitely lost: 400 bytes in 1 blocks
==962==    indirectly lost: 0 bytes in 0 blocks
==962==      possibly lost: 0 bytes in 0 blocks
==962==    still reachable: 0 bytes in 0 blocks
==962==         suppressed: 0 bytes in 0 blocks
==962==
==962== For lists of detected and suppressed errors, rerun with: -s
==962== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

free() kullanmadığımız için bellek sızıntısı yaşıyoruz

5.malloc kullanarak 100 boyutunda veri adı verilen bir tamsayı dizisi oluşturan bir program yazın; ardından data[100]'ü sıfır olarak ayarlayın. Bu programı çalıştırdığınızda ne olur? Bu programı valgrind kullanarak çalıştırdığınızda ne olur? Program doğru mu?

```
==101== Memcheck, a memory error detector
==101== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==101== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==101== Command: ./malloc2
==101==
==101== Invalid write of size 4
==101==    at 0x10916D: main (in /home/toker/malloc2)
==101==  Address 0x4a901d0 is 0 bytes after a block of size 400 alloc'd
==101==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==101==    by 0x10915E: main (in /home/toker/malloc2)
==101==
==101==
==101== HEAP SUMMARY:
==101==     in use at exit: 400 bytes in 1 blocks
==101==   total heap usage: 1 allocs, 0 frees, 400 bytes allocated
==101==
==101== 400 bytes in 1 blocks are definitely lost in loss record 1 of 1
==101==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==101==    by 0x10915E: main (in /home/toker/malloc2)
==101==
==101== LEAK SUMMARY:
==101==    definitely lost: 400 bytes in 1 blocks
==101==    indirectly lost: 0 bytes in 0 blocks
==101==      possibly lost: 0 bytes in 0 blocks
==101==    still reachable: 0 bytes in 0 blocks
==101==         suppressed: 0 bytes in 0 blocks
==101==
==101== For lists of detected and suppressed errors, rerun with: -s
==101== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

free() kullanmadığımız için yine bellek sızıntısı yaşıyoruz ve data[100] sınır dışı olduğu için invalid write işlemi yapmış olduk

6.Bir tamsayı dizisi tahsis eden (yukarıdaki gibi), onları serbest bırakan ve ardından dizinin öğelerinden birinin değerini yazdırmaya çalışan bir program oluşturun. Program çalışıyor mu? Üzerinde valgrind kullandığınızda ne olur?

```
==121== Memcheck, a memory error detector
==121== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==121== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==121== Command: ./malloc2
==121==
==121== Invalid read of size 4
==121==    at 0x109193: main (malloc2.c:6)
==121==  Address 0x4a90040 is 0 bytes inside a block of size 400 free'd
==121==    at 0x484B27F: free (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==121==    by 0x10918E: main (malloc2.c:5)
==121==  Block was alloc'd at
==121==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==121==    by 0x10917E: main (malloc2.c:4)
==121==
==121==
==121== HEAP SUMMARY:
==121==     in use at exit: 0 bytes in 0 blocks
==121==   total heap usage: 1 allocs, 1 frees, 400 bytes allocated
==121==
==121== All heap blocks were freed -- no leaks are possible
==121==
==121== For lists of detected and suppressed errors, rerun with: -s
==121== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Bu sefer free() kullandığımız için hiç bellek sızıntımız yok ancak geçersiz bir okuma işlemi yapmış bulunuyoruz

7.Şimdi serbest olarak komik bir değer iletin (örneğin, yukarıda ayırdığınız dizinin ortasındaki bir işaretçi). Ne oluyor? Bu tür bir sorunu bulmak içinaraçlara ihtiyacınız var mı?

```
malloc2.c: In function ‘int main()’:
malloc2.c:5:20: error: invalid conversion from ‘int’ to ‘void*’ [-fpermissive]
    5 |         free(data[5]);
      |              ~~~~~~^
      |                    |
      |                    int
In file included from /usr/include/c++/11/cstdlib:75,
                 from /usr/include/c++/11/stdlib.h:36,
                 from malloc2.c:2:
/usr/include/stdlib.h:555:25: note:   initializing argument 1 of ‘void free(void*)’
  555 | extern void free (void *__ptr) __THROW;
      |                   ~~~~~~~^~~~~
```

Bu sorunu bulmak için bi programa ihtiyacımız yok çünkü derleyici bize hata veriyor

8.Bellek tahsisi için diğer bazı arabirimleri deneyin. Örneğin, vektörü yönetmek için realloc() kullanan basit bir vektör benzeri veri yapısı ve ilgili rutinler oluşturun. Vektör öğelerini depolamak için bir dizi kullanın; Bir kullanıcı vektöre bir giriş eklediğinde, buna daha fazla yer ayırmak için realloc()'u kullanın. Böyle bir vektör ne kadar iyi performans gösterir? Bağlantılı bir listeyle nasıl karşılaştırılır? Hataları bulmanıza yardımcı olması için valgrind kullanın.

```
==178== Memcheck, a memory error detector
==178== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==178== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==178== Command: ./vector
==178==
0123456789
==178==
==178== HEAP SUMMARY:
==178==     in use at exit: 0 bytes in 0 blocks
==178==   total heap usage: 11 allocs, 11 frees, 1,244 bytes allocated
==178==
==178== All heap blocks were freed -- no leaks are possible
==178==
==178== For lists of detected and suppressed errors, rerun with: -s
==178== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Metotları düzgün implemente ettiğimiz takdirde bellek sızıntımız olmuyor ve yeniden bellek tahsis ederek istediğimiz depolamayı yapabiliyoruz