14

Interlude: Memory API

Bu aralarda UNIX sistemlerindeki bellek ayırma arabirimlerini ele alıyoruz. Sağlanan arabirimler oldukça basittir ve bu nedenle bölüm kısa ve nokta[1]dir. Ele aldık başlıca sorun şu:

PÜF NOKTASI : BELLEK AYIRMA VE YÖNETME
UNIX/C programlarında, belleğin nasıl paylaştırılacağını ve yönetileceğini anlamak, sağlam ve güvenilir yazılımlar oluşturmak için çok önemlidir. Yaygın olarak hangi arabirimler kullanılır? Hangi hatalardan kaçınılmalıdır?

14.1 Bellek Türler

C programını çalıştırırken, ayrılmış iki tür bellek vardır. Birincisi **yığın(stack)** belleği olarak adlandırılır ve bunların tahsisi ve atamaları sizin için derleyici, programcı tarafından dolaylı olarak yönetilir; bu nedenle bazen **otomatik (automatic)** bellek adı verilir.

C'de yığındaki belleğin bildirmesi kolaydır. Örneğin, x adı verilen bir tamsayı için işlevinde `func()` biraz alana ihtiyacınız olduğunu söyleyelim. Böyle bir bellek parçasını beyan etmek için şöyle bir şey yaparsınız:

```
void func() {
  int x; // declares an integer on the stack
  ...
}
```

Derleyici gerisini yapar, böylece `func()` çağırdığınızda yığında yer açtığınızdan emin olur. Fonksiyondan döndüğünüzde derleyici, hafızanın sizin için ayırımlarını geri alır; bu nedenle, bazı bilgilerin çağrı çağrının dışında yaşamasını istiyorsanız, bu bilgileri yığın üzerinde bırakmamanız daha iyi olurdu.

[1]Indeed, we hope all chapters are! But this one is shorter and pointier, we think.

Tüm ayırma ve ayırma işleminin programcı olarak sizin tarafınız tarafından açıkça ele alındığı öbek belleği adı verilen ikinci bellek türüne ulaşmamızı sağlamak için uzun ömürlü bir bellek gerekir. Hiç kuşku yok, ağır bir sorumluluk! Ve elbette birçok sorunun nedeni. Ancak dikkatli ve dikkatli olmanız durumunda, bu tür arayüzleri çok fazla sorun olmadan doğru şekilde kullanacaksınız. İşte, öbek üzerinde bir tamsayı nasıl atayabileceğine bir örnek:

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

Bu küçük kod parçacığıyla ilgili birkaç not. Öncelikle, bu satırda hem yığın hem de öbek atamasının gerçekleştiğini fark edebilirsiniz: Önce derleyici,

söz konusu işaretleyiciyi (`int *x`) gördüğünde bir tamsayının işaretçisi için yer açmayı bilir; daha sonra, program `malloc()` çağrıldığında, öbek üzerinde bir tamsayı için yer ister; Rutin, bu tür bir tamsayı (başarı üzerine veya başarısız olduğunda NULL) adresini geri döndürür ve bu adres daha sonra program tarafından kullanılmak üzere yığında saklanır

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

14.2 Malloc() Çağrısı

MALLOC() çağrısı oldukça basittir: Bunu bir büyüklükten geçirip yığında bir yer istemeniz gerekir ve başarılı olur ve size yeni ayrılan alana bir işaretçi verir ya da başarısız olur ve `NULL`[2]`'yi` döndürür.

Kılavuz sayfası, malloc kullanmak için yapmanız gerekeni gösterir; komut satırına `man malloc` yazın ve şunları göreceksiniz:

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

Bu bilgilerden, malloc kullanmak için yapmanız gereken tek şeyin başlık dosyası `stdlib.h'yi` içerdiğini görebilirsiniz. Aslında, tüm C programlarının varsayılan olarak birbirine bağlı olduğu C kitaplığı içinde `malloc ()` kodu olduğundan bunu yapmanız gerekmez; başlığı eklemek, derleyicinin `malloc ()'ı` doğru şekilde aradığınızı kontrol edebilmesini sağlar (örn. doğru sayıda bağımsız değişkeni doğru türde aktarıp aktarmadığınızı).

Tek parametreli `malloc ()`, sadece kaç bayta ihtiyacınız olduğunu açıklayan t tipi boyutudur. Ancak çoğu programcı buraya doğrudan bir sayı girmez (örneğin 10); gerçekten de bunu yapmak kötü bir form olarak kabul edilir. Bunun yerine, çeşitli rutinler ve makrolar

[2]Note that NULLin C isn't really anything special at all, just a macro for the value zero.

İPUCU: ŞÜPHEYE DÜŞTÜĞÜNÜZ DURUMLARDA DENEYIN
Bazı rutin veya operatörün ne şekilde davrandığından emin olamıyorsanız bunu denemenin ve beklediğiniz gibi davrandığından emin olmamanın yerini hiçbir şey alamaz. El kitabı sayfalarını veya diğer belgeleri okurken faydalıdır, uygulamada nasıl çalıştığı önemli şeydir. Biraz kod yazın ve test edin! Bu, kodunuzun istediğiniz gibi davranmasını sağlamak için en iyi yoldur. Gerçekten de, `sizeof()` hakkında söylediğimiz şeyleri iki kez kontrol etmek için yaptığımız şey doğruydu!

kullanıldı. Örneğin, çift hassasiyetli kayan nokta değeri için alan ayırmak için yapmanız gereken tek şey şu:

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

Vay canına, çok fazla şey var! Bu `malloc()` geçersiz sayma, doğru miktarda alan istemek için () operatörünü kullanır; C'de bu genellikle *bir derleme zamanı* işleci olarak düşünülür, yani gerçek boyut *derleme zamanında bilinir* ve dolayısıyla bir sayı (bu örnekte, çift için 8) `malloc ()` argümanı olarak kullanılır. Bu nedenle, `sizeof()` operatör olarak doğru şekilde düşünülmüş ve bir fonksiyon çağrısı değil (çalışma zamanında fonksiyon çağrısı yapılır)

Ayrıca, bir değişkenin adını (ve yalnızca bir türü değil) `sizeof()` olarak da geçirebilirsiniz, ancak bazı durumlarda istediğiniz sonuçları alamayabilirsiniz, bu nedenle Dikkatli olun. Örneğin, aşağıdaki kod parçacığını inceleyelim:

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

İlk satırda, iyi ve sıkıcı olan 10 tamsayı dizisi için alan ilan ettik. Ancak, sonraki satırda `sizeof()` değeri kullandığımızda, 4 (32 bit makinelerde) veya 8 (64 bit makinelerde) gibi küçük bir değer getirir. Bunun nedeni, `sizeof()`, dinamik olarak ne kadar bellek ayırdığımızı değil, tamsayı için ne kadar büyük bir *işaretçi* olduğumuzu sorduğumu düşünmesi. Ancak, `sizeof()` olması beklediğiniz gibi işe yarar:

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

Bu durumda, derleyicinin 40 bayta atanmış olduğunu bilmesi için yeterli statik bilgi vardır.

Dikkat etmek için başka bir yer ise iplerle. Bir dize için boşluk ilan ederken, şu programlama kavramını kullanın: `Malloc (strlen(s) + 1)`, bu işlem işlev `strlen()` kullanarak dize uzunluğunu alır ve dize sonu karakterine yer açmak için 1 ekler. `sizeof()` burada soruna neden olabilir.

`Malloc ()` yazmasının geçersiz olması için bir işaretçi döndürdüğünü de fark edebilirsiniz. Bu, C'de bir adresi geri vermek ve programcının bu adres ile ne yapacağına karar vermesi için tek yoldur. Programcı, bir kalıp içinde bir kodu kullanarak daha da yardımcı olur; yukarıdaki örneğimizde, programcı `malloc ()` dönüş türünü bir işaretçiye `iki katına çıkarır`. Döküm, kodunuzu okuyabilecek derleyici ve diğer programcılara söylemek dışında hiçbir şeyi başaramaz: "Evet, ne yaptığımı biliyorum." `Malloc ()` sonucunu dökerek, programcı sadece bir miktar güvence veriyor; doğruluğun doğruluğu için yayın gerekmez.

14.3 `Kullanıma Açık ()` Çağrılar

Ortaya çıktıkça, belleğin paylaştırması denklemin kolay bir parçasıdır; belleğin ne zaman, nasıl ve hatta ne zaman serbest kalacağını bilmek zor bir parçadır. Artık kullanımda olmayan öbek belleğini serbest bırakmak için programcılar **():**

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

Rutin bir bağımsız değişken alır; `malloc()` tarafından döndürülen bir işaretçi. Bu nedenle, ayrılan bölgenin boyutu kullanıcı tarafından geçmez ve bellek ayırma kitaplığının kendisi tarafından izlenmelidir.

14.4 Yaygın Hatalar

`Malloc ()` ve `free()` kullanımında ortaya çıkan bir dizi genel hata vardır. Lisans işletim sistemleri eğitimini tekrar tekrar öğrendik. Bu örneklerin tümü derleyici ile derlenir ve derleyici arasında nary a bip sesi ile çalışır; doğru bir C programı oluşturmak için C programı derlenirken, öğrendiğiniz kadar (genellikle zor yoldan) yeterli değildir.

Doğru bellek yönetimi, daha yeni birçok dilin **otomatik bellek yönetimi** (**automatic memory management)** için destek verdiği bir sorun olmuştur. Bu tür dillerde, `malloc()`'a benzer bir şey çağırıp belleği ayırırken (genellikle **yeni (new)** veya yeni bir nesne atayacak bir şey) boş alan açmak için bir şey aramanız gerekmez; bunun yerine, **çöp veri toplayıcı (garbage collector)** çalışır ve artık hangi belleğe başvuruda bulunulmadığını öğrenerek sizin için kurtarır.

Bellek Atamayı Unutma

Birçok rutin, siz aramadan önce belleğin tahsis edilmesi beklenir. Örneğin, rutin `strcpy(dst, src)` bir dizeyi kaynak işaretçisinden hedef işaretçiye kopyalar. Ancak dikkatli olmazsanız şunları yapabilirsiniz:

```
char *src = "hello";
char *dst;        // oops! unallocated
strcpy(dst, src); // segfault and die
```

İPUCU: DERLENMIŞ VEYA ÇALIŞMIŞ / = DOĞRU

Çünkü bir program derlendi (!) ya da bir ya da birçok kez doğru çalışsa bile programın doğru olduğu anlamına gelmez. Birçok etkinlik, işe yaradığını inandığın bir noktaya ulaşman için biraz ürktük olabilir, ama sonra bazı değişiklikler olur ve durur. Öğrencilerin en sık karşılaştıkları tepki "ama daha önce de işe yaradı!" demek (ya da bağırmak). ardından derleyici, işletim sistemi, donanım ve hatta profesörü suçluyoruz. Ancak sorun genellikle kodunuzda doğru olduğunu düşündüğünüz yerdir. Diğer bileşenleri suçlamadan önce işe koyup hata ayıklayın.

When Bu kodu çalıştırdığınızda, büyük ihtimalle **bölütleme aksamasına**[3] **(segmentation fault)** sebep olacaktır Kİ BU SİZİN İÇİN MAKUL BİR TERİMDİR BELLEKLE İLGİLİ YANLIŞ BİR ŞEY YAPTIN SENİ APTAL PROGRAMCI VE KIZGINIM/KIZDIM

Bu durumda, uygun kod bunun yerine şu şekilde görünebilir:

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src) + 1);
strcpy(dst, src); // work properly
```

Alternatif olarak, `strdup()` kullanabilir ve hayatınızı daha da kolaylaştırabilirsiniz.

Daha fazla bilgi için `strdup` sayfasını okuyun.

Tahsis Edilen Bellek Yeterli Değil

İlgili bir hata, bazen **arabellek taşması (buffer overflow)** olarak adlandırılan yeterli bellek ayırmıyor. Yukarıdaki örnekte, genel bir hata hedef arabelleğe neredeyse yeterli alan koymaktır.

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src)); // too small!
strcpy(dst, src); // work properly
```

İşin garip yanı, malloc'un nasıl uygulandığına ve diğer birçok ayrıntıya bağlı olarak, bu program genellikle düzgün şekilde çalışır. Bazı durumlarda, dize kopyası yürütüldüğünde, ayrılmış alanın sonuna çok fazla bir bayt yazar, ancak bazı durumlarda bu zararsızdır, belki de artık kullanılmayan bir değişkenin üzerine yazabilir. Bazı durumlarda bu aşırı akışlar inanılmaz derecede zararlı olabilir ve aslında sistemlerdeki birçok güvenlik açığına neden olabilir[W06]. Diğer durumlarda, malloc kütüphanesi bir miktar ekstra alan ayırdı ve bu nedenle programınız başka bir değişkenin değerini yazmıyor ve oldukça iyi çalışıyor. Diğer durumlarda bile, program gerçekten hata verecek ve çökecektir. Ve bu nedenle başka bir değerli ders daha öğreniriz: Bir kez doğru çalışsa da doğru olduğu anlamına gelmez.

[3]Although it sounds arcane, you will soon learn why such an illegal memory access is called a segmentation fault; if that isn't incentive to read on, what is?

Ayrılmış Belleği Başlatmayı Unutma

Bu hata ile `malloc()`'ı doğru bir şekilde çağırıyorsunuz ancak yeni atanan veri türünüze bazı değerleri girmeyi unutuyorsunuz. Bunu yapmayın! Unutursanız, programınız sonunda **başlatılmamış bir okuma (uninitialized read)** ile karşılaşacak ve burada bilinmeyen değere sahip bazı veriler yığınından okunacaktır. Orada neler olabileceğini kim biliyor? Şanslıysanız, program hala çalışır durumda (örn. sıfır) gibi bir değer elde edersiniz. Şanslıysanız rastgele ve zararlı bir şey var.

Boş Belleğe Almayı Unutma

Bir diğer yaygın hata da **bellek sızıntısı (memory leak)** olarak bilinir ve belleği serbest bırakmayı unuttuğunuz zaman meydana gelir. Uzun süre çalışan uygulamalarda veya sistemlerde (işletim sisteminin kendisi gibi), yavaş yavaş sızan bellek bir kişinin sonunda belleği tükenmesine neden olduğundan bu büyük bir sorundur ve bu noktada yeniden başlatma gerekir. Bu nedenle, genel olarak bir bellek parçasıyla işiniz bittiğinde, bunu serbest bıraktığınızdan emin olmalısınız. Çöp toplama dilinin kullanılmasının burada işe yaramadığını unutmayın: Hala bir parça belleğe başvurmanız durumunda, hiçbir çöp toplayıcı onu serbest bırakmaz ve bu nedenle bellek sızıntıları daha modern dillerde bile sorun yaratmaz.

Bazı durumlarda, `free()`'yi çağırmamak makul görünebilir. Örneğin, programınız kısa süreli ve yakında sonlandırılacak; bu durumda, işlem bittiğinde işletim sistemi tüm ayrılmış sayfaları temizleyecek ve dolayısıyla hiçbir bellek sızıntısı gerçekleşmeyecektir. Bu kesinlikle "işe yarar" (7. Sayfadaki kenara bakın) olsa da, geliştirilmesi büyük olasılıkla kötü bir alışkanlıktır, bu nedenle bu tür bir strateji seçmeye karşı dikkatli olun. Uzun vadede, programcı olarak hedeflerinden biri iyi alışkanlıklar geliştirmektir; bu alışkanlıklardan biri belleği nasıl yönettiğinizi anlamak ve (C gibi dillerde) ayırdığınız blokları serbest bırakmaktır. Bunu yapmamakla kurtulabilecek olsanız bile, açık bir şekilde atamış olduğu her bir baytı serbest bırakmak büyük olasılıkla iyi bir davranıştır.

İş Bitmeden Önce Belleği Boşaltma

Bazen bir program, belleği kullanmayı bitirmeden önce bellekte yer boşaltır; bu tür bir hataya sarkan bir işaretçi adı verilir ve tahmin edebildiğiniz gibi bu da kötü bir şeydir. Bir sonraki kullanım programı çalabilir veya geçerli belleğin üzerine yazabilir (örneğin, `free()`'yi çağırdınız, ancak daha sonra başka bir şeyi tahsis etmek için tekrar `malloc()` çağırdınız, bu da rastgele serbest kalan belleği geri dönüştürüyor)

Belleğin Sürekli Olarak Serbest Bırakılması

Programlar bazen birden fazla bellek boşaltır; bu, **çift boşaltma (double free)** olarak bilinir. Bunun sonucu tanımlanmamıştır. Tahmin edebileceğiniz gibi, bellek ayırma kitaplığı karışabilir ve her türlü tuhaf şeyi yapabilir; çökme yaygın bir sonuçtur.

BİR YANA: İŞLEMINIZ BITTIKTEN SONRA NEDEN BELLEK SIZDIRILMIYOR

Kısa ömürlü bir program yazarken malloc () kullanarak biraz alan ayırabilirsiniz. Program yürütüyor ve tamamlanmak üzere: Çıkmadan hemen önce birkaç kez ücretsiz () telefon görüşmesi gerekiyor mu? Yanlış görünse de, hiçbir bellek gerçek anlamda "kaybolmaz". Bunun nedeni çok basit: Sistemde gerçekten iki bellek yönetimi düzeyi vardır. Bellek yönetiminin ilk seviyesi, işletim sistemi tarafından gerçekleştirilir. Bu işletim sistemi, belleği çalışırken işler ve işlemler bittiğinde geri alır (veya başka bir şekilde ölür). İkinci yönetim düzeyi, örneğin malloc () ve ücretsiz () olarak aradığınızda, her süreç içinde yer alır. Program çalışmayı bitirdiğinde, boşta () işlemini arayamazsanız bile(ve böylece öbekte bellek sızdırmazsanız), işletim sistemi işlemin tüm belleğini (kod, yığın ve ilgili olan bu sayfalar da dahil olmak üzere) geri alır. Adres alanınızdaki yığının durumu ne olursa olsun, işletim sistemi işlem bittiğinde bu sayfaların tümünü geri alır ve böylece boşta bırakmamanıza rağmen bellek kaybı olmaz.

Bu nedenle, kısa ömürlü programlarda, sızdıran bellek genellikle herhangi bir çalışma problemine neden olmaz (zayıf form olarak kabul edilebilmesine rağmen). Uzun süre çalışan bir sunucu (web sunucusu veya asla kapanmayan veritabanı yönetim sistemi gibi) yazdığınızda, sızan bellek çok daha büyük bir sorundur ve eninde sonunda uygulamanın belleği bittiğinde bir çökmeye neden olur. Ve elbette, bellek sızıntısı belirli bir programda daha da büyük bir sorundur: işletim sisteminin kendisi. Bu bize bir kez daha gösteriyor ki: çekirdek kodunu yazanların işi en zor olanlardır...

`free()`'yi Yanlış Arama

Üzerinde duracağımız son bir konu Free()'yi yanlış aramak. 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 tehlikelidir ve elbette bundan da kaçınılmalıdır.

Özet

Gördüğünüz gibi, belleği kötüye kullanmanın birçok yolu vardır. Bellekteki sık sık ortaya çıkan hatalar nedeniyle, kodunuzda bu tür sorunları bulmanıza yardımcı olmak için bir araç küresi geliştirilmiştir. Hem temizlik [HJ92] hem de valgrind [SN05] olup olmadığını kontrol edin; her ikisi de bellekle ilgili sorunlarınızın kaynağını bulmanıza yardımcı olmak için mükemmeldir. Bu güçlü araçları kullanmaya alıştıktan sonra, onlar olmadan 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, kitaplık çağrılarıdır. Böylece, malloc kitaplığı, sanal adres alanınızdaki alanı yönetir, ancak kendisi, daha fazla bellek istemek veya bir kısmını sisteme geri bırakmak için işletim sistemine çağrı yapan bazı sistem çağrılarının üzerine kuruludur.

Bu tür bir sistem çağrısı, program kopmasının konumunu değiştirmek için kullanılan brk olarak adlandırılır: yığın bitişinin konumu. Bir bağımsız değişken (yeni kopmanın adresi) gerektirir ve dolayısıyla yeni kopmanın geçerli kopuktan daha büyük veya daha küçük olup olmamasına bağlı olarak yığın boyutunu artırır veya azaltır. Ek bir çağrı sbrk bir artış gösterir ancak bunun dışında benzer bir amaca hizmet eder.

Asla doğrudan ya sbrk'ı aramamanız gerektiğini unutmayın. Bunlar bellek ayırma kitaplığı tarafından kullanılır; onları kullanmaya çalışırsanız, muhtemelen bir şeylerin (korkunç bir biçimde) ters gitmesine neden olursunuz. Bunun yerine "malloc ()" ve "free ()" ye bağlı kalın.

Son olarak "mmap()" çağrısı ile işletim sisteminden de bellek alabilirsiniz. Doğru bağımsız değişkenleri ileterek, "mmap ()" programınızda anonim bir bellek bölgesi oluşturabilir. Bu alan, herhangi bir dosyayla değil, takas alanıyla ilişkili olan bir alandır ve daha sonra sanal bellekte ayrıntılı olarak ele alacağız. Bu bellek daha sonra bir yığın gibi ele alınabilir ve bu şekilde yönetilebilir. Daha fazla ayrıntı için "mmap ()" kılavuz sayfasını okuyun.

14.6 Diğer Çağrılar

Bellek ayırma kitaplığının desteklediği birkaç başka çağrı vardır. Örneğin, `malloc()` belleği ayırır ve geri dönmeden önce onu sıfırlar; bu, belleğin sıfırlandığını varsaydığınız ve belleği kendiniz başlatmayı unuttuğunuz bazı hataları önler (yukarıdaki "başlatılmamış okumalar" ile ilgili paragrafa bakın). `Realloc()` program parçası, 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 ayırmayla ilgili bazı API'leri kullanıma sunduk. Her zamanki gibi temel konuları ele aldık; başka yerlerde daha fazla ayrıntı bulabilirsiniz. Daha fazla bilgi için C kitabını [KR88] ve Stevens [SR05] (Bölüm 7) okuyun. Bu sorunların çoğunun otomatik olarak nasıl algılanıp düzeltileceğini anlatan havalı modern bir makale için, bkz. Novark vd. [N+07]; bu makale ayrıca yaygın sorunların güzel bir özetini ve bunların nasıl bulunup düzeltileceğine dair bazı güzel fikirler içermektedir.

References

[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. Available: 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 tahsisi konusunda biraz bilgi sahibi olacaksınız. Öncelikle bazı hatalı programları yazacaksınız (komik!). Daha sonra yerleştirdiğiniz hataları bulmanıza yardımcı olacak bazı araçlar kullanacaksınız. Ardından bu araçların ne kadar muhteşem olduğunu fark edeceksiniz ve gelecekte nasıl kullanacaksınız, böylece kendinizi daha mutlu ve üretken hissedeceksiniz. Hata ayıklayıcı (örn. gdb) ve bir bellek hatası dedektörü olan araçlar `valgrind` olarak adlandırılır. [SN05].

Sorular

1. İlk olarak, tamsayı için bir işaretçi oluşturan, onu NULL olarak ayarlayan ve sonra onu referanstan çıkarmaya çalışan `null.c` adlı basit bir program yazın. Bunu null adı verilen çalıştırılabilir bir bilgisyar programına derleyin. Bu programı çalıştırdığınızda ne olur?

```c
int main(void)
{
  // Create a pointer to an integer
  int *pointer = NULL;

  // Set the pointer to NULL
  pointer = NULL;

  // Try to dereference the pointer
  *pointer = 5;

  return 0;
}
```

Programı çalıştırdığınızda, program bir NULL işaretçisinin başvurusunu kaldırmaya çalıştığı için muhtemelen çökecek veya bir hata üretecektir. Bir NULL işaretçisinin başvurusunu kaldırmak, işaretçinin işaret ettiği verilere erişmeye çalışmak anlamına gelir, ancak işaretçiye herhangi bir geçerli bellek adresi atanmadığı için bu işleme izin verilmez ve programın çökmesine neden olabilir. Bu tür hataları önlemek için kodunuzdaki NULL işaretçilerinin başvurusunu kaldırmaktan kaçınmak genellikle en iyi uygulama olarak kabul edilir.

Programın kodu şöyle görünebilir:

```
gcc -o null null.c
```

2. Daha sonra, bu programı simge bilgileri ile derleyin (g işareti). Bunu yaparak, hata ayıklayıcı'nın değişken adları ve benzeri hakkında daha faydalı bilgilere erişmesini sağlamak için yürütülebilir dosyaya daha fazla bilgi yerleştirelim. Programı null yazarak hata ayıklayıcı adı altında çalıştırın ve daha sonra `GDB` programı verileri karşılaştırırken çalıştır yazın. `GDB` size ne gösteriyor?

   Programı sembol bilgilerini içerecek şekilde -g bayrağıyla derler ve ardından hata ayıklayıcı altında çalıştırırsanız, gdb size programın kilitlendiği kod satırını ve ilgili hata mesajını gösterir. Ayrıca, çöken kod satırı kapsamındaki tüm yerel değişkenlerin değerlerini de gösterecektir. Bu bilgi, hatanın nedenini belirlemede ve programda hata ayıklamada yararlı olabilir.

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000004004e7 in main () at null.c:12
12          *pointer = 5;
(gdb) print pointer
$1 = (int *) 0x0
```

3. Son olarak bu programdaki `valgrind` destek programını kullanın. Ne olduğunu analiz etmek için `valgrind` programının bir parçası olan memcheck destek programını kullanacağız. Şunu yazarak çalıştırın: `valgrind --leak-check=yes null`. Çalıştırdığınızda ne oluyor? Destek programından çıkan şeyi yorumlayabilir misiniz?

   Bu komutu çalıştırdığınızda valgrind, programın bellek kullanımını analiz edecek ve bulduğu hataları veya sızıntıları rapor edecektir. Bu durumda, valgrind muhtemelen aşağıdakine benzer bir hata bildirecektir:

```
==3210== Invalid read of size 4
==3210==    at 0x4004E7: main (null.c:12)
==3210==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

4. `Malloc()` kullanarak bellek tahsis eden basit bir program yazınancak çıkış yapmadan önce kullanıma açık olmayı unutur. Bu program çalışırken ne oluyor? `GDB`'yi bu programla ilgili herhangi bir sorun bulmak iççin kullanabilir misiniz? `Valgrind`'de nasıl bir işlem olur?

```
==2678== ...
==2678== HEAP SUMMARY:
==2678==     in use at exit: 4 bytes in 1 blocks
==2678==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==2678==
==2678== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2678==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2678==    by 0x400545: main (forgetful_alloc.c:5)
==2678==
==2678== LEAK SUMMARY:
==2678==    definitely lost: 4 bytes in 1 blocks
==2678==    indirectly lost: 0 bytes in 0 blocks
==2678==      possibly lost: 0 bytes in 0 blocks
==2678==    still reachable: 0 bytes in 0 blocks
==2678==         suppressed: 0 bytes in 0 blocks
==2678== ...
```

Bir program malloc() kullanarak belleği ayırırsa ancak çıkmadan önce belleği boşaltmayı unutursa, bellek işletim sistemine geri bırakılmaz. Bu, programın serbest bırakmadan zaman içinde artan miktarda bellek kullandığı bir bellek sızıntısına neden olabilir. Sonunda, programın belleği tükenebilir ve çökebilir veya sistemdeki diğer programların da belleğinin bitmesine ve çökmesine neden olabilir. Gdb size değişkenlerin değerlerini ve belleğin içeriğini gösterebilir ve programla ilgili bellek sızıntılarını veya diğer sorunları belirlemenize yardımcı olabilir. Valgrind ile bir program çalıştırdığınızda, programın bellek kullanımını izler ve bulduğu sorunları bildirir.

5. `Malloc()` kullanarak veri olarak bilinen boyutu 100 olan bir tam sayı dizisi oluşturan bir program yazın; ardından veriyi 100'den 0'a ayarlayın. Bu programı çalıştırdığınızda ne oluyor? `Valgrind` kullanarak bu programı çalıştırdığınızda ne oluyor? Program doğru mu?

```
==2598== ...
==2598== Invalid write of size 4
==2598==    at 0x400554: main (int_array.c:6)
==2598==  Address 0x51fc1d0 is 0 bytes after a block of size 400 alloc'd
==2598==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2598==    by 0x400545: main (int_array.c:5)
==2598==
==2598==
==2598== HEAP SUMMARY:
==2598==     in use at exit: 400 bytes in 1 blocks
==2598==   total heap usage: 1 allocs, 0 frees, 400 bytes allocated
==2598==
==2598== 400 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2598==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2598==    by 0x400545: main (int_array.c:5)
==2598==
==2598== LEAK SUMMARY:
==2598==    definitely lost: 400 bytes in 1 blocks
==2598==    indirectly lost: 0 bytes in 0 blocks
==2598==      possibly lost: 0 bytes in 0 blocks
==2598==    still reachable: 0 bytes in 0 blocks
==2598==         suppressed: 0 bytes in 0 blocks
==2598== ...
```

program muhtemelen çökecek veya beklenmeyen sonuçlar üretecektir. Bunun nedeni, malloc() tarafından ayrılan dizinin yalnızca 100 öğelik alana sahip olması ve data[100]'e erişmeye çalışmanın programın ayrılan dizinin dışındaki belleğe erişmesine neden olmasıdır. Bu programı valgrind ile çalıştırdığınızda, valgrind muhtemelen arabellek taşmasını algılayacak ve bunu bir hata olarak bildirecektir. Genel olarak, programın çökmesine veya beklenmeyen sonuçlara neden olabilecek bir arabellek taşması içerdiğinden, program doğru değildir.

6. Bir dizi tamsayı (yukarıdaki gibi) tahsis eden, serbest bırakmayı sağlayan ve ardından dizi öğelerinden birinin değerini yazdırmaya çalışan bir program oluşturun. Program çalışıyor mu? Programda `valgrind` kullandığınızda neler oluyor?

```
==2652== ...
==2652== Invalid read of size 4
==2652==    at 0x4005F4: main (data_freed.c:9)
==2652==  Address 0x51fc040 is 0 bytes inside a block of size 400 free'd
==2652==    at 0x4C2BDEC: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==2652==    by 0x4005EF: main (data_freed.c:8)
==2652== ...
```

Bir program bir tamsayı dizisi tahsis ederse, diziyi serbest bırakır ve ardından dizinin öğelerinden birinin değerini yazdırmaya çalışırsa, program büyük olasılıkla çökecek veya beklenmeyen sonuçlar üretecektir.Bu programı valgrind ile çalıştırdığınızda, valgrind muhtemelen geçersiz belleğe erişimi algılayacak ve bunu bir hata olarak bildirecektir.

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

Derler, ancak bir çalışma zamanı hatası vardır. Gdb kesinlikle daha büyük bir programda hata ayıklamaya yardımcı olacaktır.

8. Bellek ayırma için diğer arabirimlerden bazılarını deneyin. Örneğin, basit bir vektör benzeri veri yapısı ve rektörü kontrol etmek için relloc kullanan program parçaları oluşturun. Vektörün elementlerini depolamak için bir dizin kullanın; bir kullanıcı vektöre giriş yaptığında vektöre daha fazla alan ayırmak için `realloc()` kullanın. Bu tür bir vektör ne kadar iyi bir performans gösterir? Bağlantılı bir listeye göre nasıldır? Hataları bulmanıza yardımcı olması için `valgrind` kullanın.

   Genel olarak, belleği yönetmek için realloc() kullanan bir vektör, vektördeki öğelerin sayısı arttığında iyi performans gösterir, çünkü realloc() gerektiği gibi daha büyük bellek bloklarını verimli bir şekilde ayırabilir. Bağlantılı bir liste, her öğe eklendiğinde veya çıkarıldığında tüm listeyi kopyalaması gerekmediğinden, öğelerin daha verimli bir şekilde eklenmesini ve silinmesini sağlayabilir. Bununla birlikte, bağlantılı bir listedeki her öğe, listedeki bir sonraki öğeye bir işaretçi depolamak zorunda olduğundan, bağlantılı bir liste, bir vektöre göre öğe başına daha fazla bellek ek yükü gerektirir.

9. `GDB` ve `valgrind` kullanımı hakkında daha fazla zaman harcayın ve bilgi edinin. Araçlarınızı bilmek son derece önemlidir; UNIX ve C ortamında uzman hata ayıklayıcı olmak için zaman ayırın ve nasıl çalışmanız gerektiğini öğrenin.