

Bölüm: Bellek Arayüzü

Bu bölümde, UNIX sistemindeki bellek paylaşırma (bölüşdürme) arayüzünü tartışacağız. Sağlanan bu arayüz oldukça basit, ve bu yüzden bölüm kısa ve öz¹. Ele aldığımız temel problem şudur:

MESELE: BELLEK NASIL YER TAHSİS EDİLİR VE YÖNETİLİR?

UNIX/C programlarında, belleğin nasıl tahsis edileceğini ve yönetileceğini anlamak, sağlam ve güvenilir yazılım oluşturmada kritik öneme sahiptir. Hangi arayüzler yaygın olarak kullanılır? Hangi hatalardan kaçınılmalıdır?

14.1 Bellek Çeşitleri

Çalıştırılan bir C programında, ayrılan iki tip bellek vardır. İlki **yığın (stack)** bellek olarak adlandırılır ve tahsis edilen ve serbest bırakılan yerler sizin için derleyici tarafından örtük olarak yönetilir, programcı; bu nedenden dolayı bazen onu **otomatik (automatic)** bellek olarak adlandırır.

C'deki yığında (stack) bellek tanımlamak kolaydır. Örneğin, x adlı bir tamsayı için func() fonksiyonunda biraz boşluğa ihtiyacınız olduğunu varsayalım. Böyle bir belleği tanımlamak için, sadece bunun gibi bir şeyler yapmalısınız:

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

Gerisini derleyici halleder, func()'ı çağırdığınızda yığında boşluk bıraktığınızdan emin olun. Fonksiyondan geri dönüşünüzde, derleyici sizin için belleği serbest bırakır; bu nedenle, bazı bilgilerin çağrı çağrısının (call

¹ Gerçekten de tüm bölümlerin aynı olmasını umuyoruz! Fakat bu bölüm daha kısa ve daha belirleyici diye düşünüyoruz.

invocation) ötesinde yaşamasını istiyorsanız, bu bilgiyi yığında bırakmazsanız iyi olur.

1

Uzun ömürlü belleğe duyulan bu ihtiyaç, bizi tüm tahsislerin ve serbest bırakmaların açıkça siz programcılar tarafından işlendiği **öbek(heap)** bellek adı verilen, ikinci tür belleğe götürür. Kuşkusuz, ağır bir sorumluluk! Ve kesinlikle birçok hatanın da nedenidir. Fakat dikkatli olursanız ve dikkat ederseniz, bu tür arayüzleri doğru ve çok fazla sorun yaşamadan kullanacaksınız. Buradaki bir sayıya(integer) nasıl bellek tahsis edilebileceğine dair bir örnektir:

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

Bu küçük kod parçacığı hakkında birkaç not: İlk olarak, bu satırda hem yığın(stack) hem de öbek (heap) tahsisinin gerçekleştiğini fark edebilirsiniz: ilk önce derleyici, söz konusu (int * x) işaretçisinin serbest bırakıldığını gördüğünde; bir sayı işaretçisi için yer açmayı bilir. Daha sonra, program malloc()'u çağırdığında, öbek(heap) üzerinde bir tam sayı için yer ister; rutini, daha sonra program tarafında kullanılmak üzere yığında depolanan bir tam sayının adresini (başarı durumunda işaretçiyi ya da başarısızlık durumunda NULL' u) döndürür.

Doğası gereği ve daha çeşitli kullanımı nedeniyle öbek bellek hem kullanıcılar hem de sistem için daha fazla zorluk sunar. Bu nedenle, tartışmamızın geri kalanının odak noktasıdır.

14.2 malloc()'un Çağırılması

malloc(), oldukça basit çağırılır: öbekte yer isteyen bir boyut iletirsiniz ve ya başarılı olur ve size yeni ayrılan alana bir işaretçi verir ya da başarısız olur ve NULL² döndürür.

Kılavuz sayfa, malloc()'u kullanabilmeniz için ne yapmanız gerektiğini gösterir. Yorum satırına man malloc yazın ve göreceksiniz:

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

Bu bilgilerden, malloc' u kullanmak için yapmanız gerekenin stdlib.h başlık dosyasını dahil etmek olduğunu görebilirsiniz. Aslına bunu bile gerçekten yapmanıza gerek yok, çünkü tüm C programlarının varsayılan olarak bağlandığı C kütüphanesi içinde malloc() koduna sahiptir. Başlığın eklenmesi sadece derleyicinin, malloc()' u doğru çağırıp çağırmadığınızı kontrol etmesine izin verir(örneğin ona doğru sayıda doğru tipten argüman iletilmesi).

`malloc()` parametresinin aldığı tek parametre, kaç byte'a ihtiyacınız olduğunu açıklayan sızet türüdür. Bununla birlikte, çoğu programcı buraya doğrudan bir sayı yazmaz (10 gibi) gerçekten bunu yapmak zayıf bir form olarak kabul edilir. Bunun yerine çeşitli rutinler ve makrolar kullanılır.

İpucu: ŞÜPHE DUYDUĞUNUZDA, TEKRAR DENEYİN

Eğer kullandığınız bir rutinin veya operatörün nasıl davrandığından emin değilseniz, onu deneyip beklediğiniz gibi davrandığından emin olmanın yerini hiçbir şey tutamaz. Kılavuz sayfalarını veya diğer belgeleri okumak yararlı olsa da önemli olan pratikte nasıl çalıştığıdır. Biraz kod yazın ve test edin! Kodunuzun istediğiniz gibi davrandığından emin olmanın en iyi yolu şüphesiz budur. Nitekim, `sizeof()` hakkında söylediklerimizin gerçekten doğru olduğunu iki kez kontrol etmek için yaptığımız şey buydu!

Örneğin, çift duyarlıklılı bir kayan nokta değerine yer ayırmak için şunu yapmanız yeterlidir:

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

Vay, çok fazla çiftleme(double-ing). `malloc()`'un bu çağrısı, doğru miktarda alan istemek için `sizeof()` operatörünü kullanır; C'de bu genellikle bir derleme zamanı operatörü olarak düşünülür, yani gerçek boyutun derleme zamanında bilindiği ve bu nedenle `malloc()` argümanının bir sayı(bu durumda, bir çift(double) için 8) yerine geçtiği anlamına gelir. Bu nedenle, `sizeof()` bir fonksiyon çağrısı olarak değil, bir operatör olarak düşünülmelidir (bir fonksiyon çağrısı çalışma zamanında gerçekleşir).

Ayrıca `sizeof()` fonksiyonuna bir değişken adını (yalnızca türünü değil) da aktarabilirsiniz fakat bazı durumlarda istediğiniz sonuçları alamayabilirsiniz, bu yüzden dikkatli olun. Örneğin, aşağıdaki kod parçasına bakalım:

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

İlk satırda, 10 tane tam sayılık bir dizi için alan bildirdik, ki bu gayet iyi. Ancak, bir sonraki satırda `sizeof()`'u kullandığımızda, 4 (32 bitlik makinelerde) ya da 8 (64 bitlik makinelerde) gibi küçük bir değer döndürür. Bunun nedeni, bu durumda `sizeof()` fonksiyonunun dinamik olarak değil, sadece bir tamsayı işaretçisinin ne kadar büyük olduğunu sordüğümüzü düşünmesidir. Ancak bazen `sizeof()` beklediğiniz gibi çalışabilir:

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

Bu durumda, derleyicinin 40 bayt yer tahsis edildiğini bilmesi için yeterli statik bilgi vardır.

Dikkat edilmesi gereken bir diğer yer de dizelerdir(string). Bir dize için alan bildirirken, şu deyiimi kullanın: `malloc(strlen(s) + 1)`, `strlen()` fonksiyonunu kullanarak dizinin uzunluğunu alır ve dizge sonu karakterine yer açmak için buna 1 ekler. Burada `sizeof()` fonksiyonunu kullanmak sorunlara yol açabilir.

Aynı zamanda `malloc()`'un void tipinde bir işaretçi döndürdüğünü de fark edebilirsiniz. Bunu yapmak, C'de bir adresi geri aktarmanın ve programcının onunla ne yapacağına karar vermesine izin vermenin tek yoludur. Ayrıca programcı, **tür değişimi(cast)** olarak adlandırılan yöntemi kullanarak yardımcı olur; yukarıdaki örneğimizde programcı, `malloc()` dönüş türünü bir double işaretçisine atar. Tür değişimi derleyiciye ve kodunuzu okuyabilecek diğer programcılara "Evet, ne yaptığımı biliyorum." demekten başka hiçbir şey yapmaz. Programcı `malloc()` sonucunun türünü değiştirerek, sadece biraz güvence verir; doğruluk için tür değişimine gerek yoktur.

14.3 free() 'nin Çağırılması

Anlaşıldığı gibi, bellek tahsis etmek denklemin kolay kısmıdır; ne zaman, nasıl ve hatta belleğin boş olup olmadığını bilmek ise zor kısmıdır. Artık kullanılmayan öbek belleğini boşaltmak için, programcılar **free()** fonksiyonunu çağırması yeterlidir:

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

Rutin olarak `malloc()` tarafından döndürülen bir işaretçi argümanı alır. Bu nedenle, tahsis edilen bölgenin boyutunun kullanıcı tarafından aktarılmadığını ve bellek tahsis kütüphanesinin kendisi tarafından takip edilmesi gerektiğini fark edebilirsiniz.

14.4 Sık Karşılaşılan Hatalar

`malloc()` ve `free()` kullanımında ortaya çıkan bazı yaygın hatalar vardır. İşte lisans işletim sistemleri dersini öğretirken tekrar tekrar gördüğümüz bazı örnekler. Bu örneklerin tümü derleyiciden hiçbir ses çıkmadan derlenir ve yürütülür; bir C programını derlemek doğru bir C programını oluşturmak için gerekli olsa da, öğreneceğiniz gibi (genellikle zor yoldan) yeterli olmaktan uzaktır.

Aslında doğru bellek yönetimi öyle bir sorun olmuştur ki birçok yeni dil **otomatik bellek yönetimini (automatic memory management)** desteklemektedir. Bu gibi dillerde, belleği ayırmak için `malloc()` benzeri bir şey çağırırken (genellikle **yeni (new)** veya yeni bir nesneye yer tahsis etmek gibi bir şey), boş alan için hiçbir zaman bir şey çağırmanız gerekmez; bunun yerine, bir **çöp toplayıcı (garbage collector)** çalışır ve artık hangi belleğe başvurmadığınızı bulur ve onu sizin için boşaltır.

Bellek Ayırmayı Unutmak

Birçok rutin siz onu çağırmadan önce bellek ayrılmasını bekler. Örneğin, `strcpy(dst, src)` rutini bir dizeyi(string) kaynak işaretçiden hedef işaretçiye atar. Ancak, dikkatli olmazsanız, bunu yapabilirsiniz:

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

İPUCU: DERLENDİ VEYA ÇALIŞTIRILDI: DOĞRU

Bir programın derlenmiş(!) olması, hatta bir ya da birçok kez doğru çalıştırılmış olması programın doğru olduğu anlamına gelmez. Birçok olay sizi çalıştığına inandığınız bir noktaya getirmek için bir araya gelmiş olabilir, ancak daha sonra bir şeyler değişir ve program durur. Yaygın bir öğrenci tepkisi “Ama daha önce çalışıyordu!” demek (veya bağırarak), derleyiciyi, işletim sistemini donanımı ve hatta (söylemeye cesaret edersek) profesörü suçlamaktır. Ancak sorun genellikle tam da düşündüğünüz yerde, kodunuzdadır. Diğer bileşenleri suçlamadan önce işe koyulun ve hata ayıklayın.

Kodunuzu çalıştırdığınızda büyük olasılıkla bir **segmentasyon hatasına²** (**segmentation fault**) yol açacaktır, bu da **SEN BELLEKTE YANLIŞ BİR ŞEY YAPTIN SEN APTAL PROGRAMCISIN VE BEN KIZGINIM** anlamına gelen süslü bir terimdir.

Bu durumda, bunun yerine uygun kod aşağıdaki gibi görünebilir:

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

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

Daha fazla bilgi için `strdup` kılavuzunu okuyabilirsiniz.

Yeterli Bellek Ayrılamıyor

Yeterli bellek ayrılamamasıyla ilgili hata, bazen **tampon taşması (buffer overflow)** olarak adlandırılır. Yukarıdaki örnekteki, hedef tampon için neredeyse yeterli yer ayırmaktadır yaygın bir hatadır.

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

Garip bir şekilde, `malloc`'un nasıl uygulandığına ve diğer birçok ayrıntıya bağlı olarak, bu program görünüşte genellikle doğru şekilde çalışacaktır. Bazı durumlarda, dize kopyası çalıştırıldığında, ayrılan alanın sonuna bir byte fazla

yazılır, ancak bazı durumlarda belki de artık kullanılmayan bir değişkenin üzerine yazıldığından bu zararsızdır. Bazı durumlarda, bu taşmalar inanılmaz derecede zararlı olabilir ve aslında sistemlerdeki birçok güvenlik açığının kaynağıdır [W06]. Diğer durumlarda, `malloc` kütüphanesi zaten fazladan biraz yer ayırmıştır ve bu nedenle programınız aslında başka bir değişkenin değerini karalamaz ve oldukça iyi çalışır. Hatta diğer durumlarda, program gerçekten hata verecek ve çökecektir. Ve böylece değerli bir ders öğrenmiş oluruz: doğru çalışmış olması doğru olduğu anlamına gelmez.

Ayrılan Belleği Başlatmayı Unutmak

Bu hata ile birlikte yeni ayrılan veri türünüze bazı değerler doldurmaya unutursunuz ancak `malloc()` işlevini doğru bir şekilde çağırırsınız, Bunu yapmayın! Eğer unutursanız, programınız eninde sonunda heap'ten değeri bilinmeyen bazı verileri okuduğundan **başlatılmamış okuma (uninitialized read)** ile karşılaşır. İçinde ne olduğunu kim bilebilir? Eğer şanslıysanız, bazı değerler programın hala çalışmasını sağlayacak (örneğin sıfır). Şanslı değilseniz, rastgele ve zararlı şeylerdir.

Belleği Boşaltmayı Unutmak

Bir başka tanınmış yaygın hata da **bellek sızıntısı (memory leak)** olarak bilinir ve belleği boşaltmayı unuttuğunuzda ortaya çıkar. Uzun süre çalışan uygulamalarda veya sistemlerde (işletim sisteminin kendisi gibi) bu büyük bir sorundur, çünkü yavaşça sızan bellek sonunda belleğin tükenmesine neden olur ve bu noktada yeniden başlatma gerekir. Bu nedenle, genelde, bir bellek yığınıyla işiniz bittiğinde, onu boşalttığınızdan emin olmalısınız. Çöp toplayan bir dil kullanmanın bu konuda yardımcı olmadığını unutmayın: bir bellek parçasına hala bir referansınız varsa, hiçbir çöp toplayıcı onu asla serbest bırakmaz ve bu nedenle bellek sızıntıları daha modern dillerde bile bir sorun olmaya devam eder.

Bazı durumlarda, `free()` fonksiyonunu çağırarak makul görülebilir. Örneğin, programınız kısa ömürlüdür ve kısa süre sonra çıkacaktır; bu durumda, süreç öldüğünde, işletim sistemi yer tahsis edilmiş tüm sayfaları temizleyecektir ve böylece herhangi bir bellek sızıntısı gerçekleşmeyecektir. Bu kesinlikle “işe yarasa” da (sayfa 7’ deki kenara bakınız), muhtemelen kötü bir alışkanlıktır, bu nedenle böyle bir strateji seçerken dikkatli olun. Uzun vadede, bir programcı olarak hedeflerinizden biri iyi alışkanlıklar geliştirmektir; bu alışkanlıklardan biri de belleği nasıl yönettiğinizi anlamak ve (C gibi dillerde) ayırdığınız bloklar serbest bırakmaktır. Bunu yapmadan kurtulabilirsiniz bile, açıkça ayırdığınız her bir baytı serbest bırakma alışkanlığı edinmek muhtemelen iyi olacaktır.

İşiniz Bitmeden Önce Belleği Boşaltmak

Bazen bir program belleği kullanmayı bitirmeden önce serbest bırakır; 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()` fonksiyonunu çağırdınız, ancak daha

sonra başka bir şey ayırmak için `malloc()` fonksiyonunu tekrar çağırdınız, bu da hatalı olarak serbest bırakılan belleği geri dönüştürür).

Belleği Tekrar Tekrar Boşaltma

Programlar bazen belleği birden fazla kez boşaltır; bu **çifte boşaltma (double free)** olarak bilinir. Bunu yapmanın sonucu tanımsızlıktır. Tahmin edebileceğiniz gibi, belleğin yere tahsisi kütüphanesi karma karışık olabilir ve her türlü garip şeyi yapabilir; çökmeler yaygın bir sonuçtur.

BİR KENAR: İŞLEMİNİZDEN ÇIKTIĞINIZDA NEDEN BELLEK SIZDIRILMIYOR?

Kısa ömürlü bir program yazarken `malloc()` kullanarak bir miktar alan ayırabilirsiniz. Program çalışıyor ve tamamlanmak üzere: Çıkmadan hemen önce `free` ögesini birkaç kez çağırmanız gerekiyor mu? Bunu yapmamak yanlış gibi görünse de gerçek anlamda hiçbir bellek “kaybolmayacaktır”. Bunun nedeni basittir: sistemde gerçekten iki seviyede bellek yönetimi vardır. Bellek yönetiminin ilk seviyesi işletim sistemi tarafından gerçekleştirilir, bu sistem belleği süreçler çalışırken onlara verir ve süreçler çıktığında (ya da başka bir şekilde öldüğünde) geri alır. Yönetimin ikinci seviyesi ise her bir süreci *içinde*, örneğin `malloc()` ve `free()` fonksiyonlarını çağırdığınızda heap içinde gerçekleşir. `free()` işlevini çağırmayı başaramasanız bile (ve dolayısıyla heap’te bellek sızıntısı olsa bile), programın çalışması bittiğinde işletim sistemi sürecin tüm belleğini (kod, stack ve burada konuyla ilgili olarak heap sayfaları dahil) geri alacaktır, adres alanınızdaki heap’ in durumu ne olursa olsun, süreç öldüğünde işletim sistemi bu sayfaların tümünü geri alır, böylece belleği boşaltmamış olmanıza rağmen hiçbir belleğin kaybolmamasını sağlar.

Böylece, kısa ömürlü programlar için, bellek sızıntısı genellikle herhangi bir operasyonel soruna neden olmaz (kötü biçim olarak kabul edilse de). Uzun süre çalışan bir sunucu yazdığınızda (web sunucusu veya veri tabanı yönetim sistemi gibi hiç çıkmayan), bellek sızıntısı çok daha büyük bir sorundur ve uygulama belleği bittiğinde eninde sonunda çökmeye yol açar. Ve elbette, bellek sızıntısı belirli bir programın içinde daha da büyük bir sorundur: İşletim sisteminin kendisi. Bize bir kez daha gösteriyor ki: çekirdek kodunu yazarların işi en zor olanıdır...

`free()` Fonksiyonunun Yanlış Çağırılması

Tartıştığımız son bir sorun da `free()` fonksiyonunun yanlış çağırılmasıdır. Sonuçta, `free()` sizden sadece daha önce `malloc()`’tan aldığınız işaretçilerden birini geçirmenizi bekler. Başka bir değer ilettiğinizde, kötü şeyler olabilir (ve oluyor). Bu nedenle, bu tür **geçersiz boşaltmalar (invalid free)** tehlikelidir ve elbette kaçınılması gerekir.

Özet

Gördüğünüz gibi, belleği kötüye kullanmanın pek çok yolu vardır. Bellekle ilgili sıkça yapılan hatalar nedeniyle, kodunuzdaki bu tür sorunları bulmanıza yardımcı olacak bir araç ekosferi gelişmiştir. Hem **arındırma (purify)** [HJ92] hem de **valgrind** [SN05] araçlarına göz atın; her ikisi de bellekle ilgili sorunlarınızın kaynağını bulmanıza yardımcı olma konusunda mükemmeldir. Bu güçlü araçları kullanmaya alıştığınızda, onlar olmadan nasıl hayatta kaldığınızı merak.

14.5 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 çağırarak bazı sistem çağrılarının üzerine inşa edilmiştir.

Programın **kesme(break)** konumunu, öbeğin 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 öbeğin boyutunu artırır veya azaltır. Ek olarak `sbrk` çağırısına bir artış(increment) aktarılır ancak bunun dışında benzer bir amaca hizmet eder.

`brk` ya da `sbrk'` yi asla doğrudan çağırmanız 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()` çağırısı aracılığıyla işletim sisteminden de bellek elde edebilirsiniz. Doğru argümanları ileterek, `mmap()` programınız içinde **anonim(anonymous)** 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 öbek olarak alınabilir ve bu şekilde yönetilebilir. Daha fazla ayrıntı için `mmap()` 'in kılavuz sayfasını okuyun.

14.6 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 işaretçisini döndürür.

14.7 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.

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.*

Ev Ödevi (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 [SN05].

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 (.exe) derleyin. Bu programı çalıştırdığınızda ne olur?

Verilen isterlere göre aşağıdaki kodu oluşturdum.

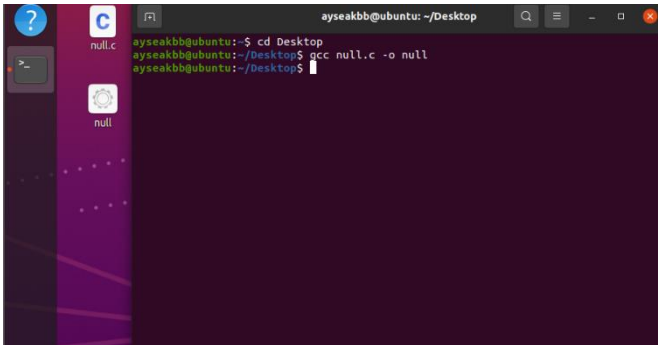
```

Open  ▾  [F]  *null.c  ~/Desktop  Save  ≡  -  □  ×

1 #include <stdio.h>
2
3 int main(void) {
4
5     int *pi, i;           //pi pointer'ı ve i değişkeni tanımlandı.
6     i = 5;                //i değişkenine değer atandı
7     pi = &i;              //i'nin adresi pointer'a aktarıldı
8     pi = NULL;            //pointer'daki adresi null yaparak değişkeni referanssız hale getirildi.
9
10
11     printf("i değeri = %d", i);    //i değerini ekrana yazdırılır.
12     printf("Pointer değeri = %d", *pi);    //Pointer değeri ekrana yazdırılır.
13
14     return 0;
15 }

```

Ubuntu 20.04 işletim sisteminde, verilen kodu terminal aracılığıyla derleyip çalıştırılabilir format olan .out formatında bir dosya oluşturdum. Dosya adı sadece `gcc null.c` yapılırsa otomatik olarak `a.out` şeklinde oluşturulur. Ben istediğim isim olan `null` ile oluşturmak için `gcc null.c -o null` kodunu kullandım.

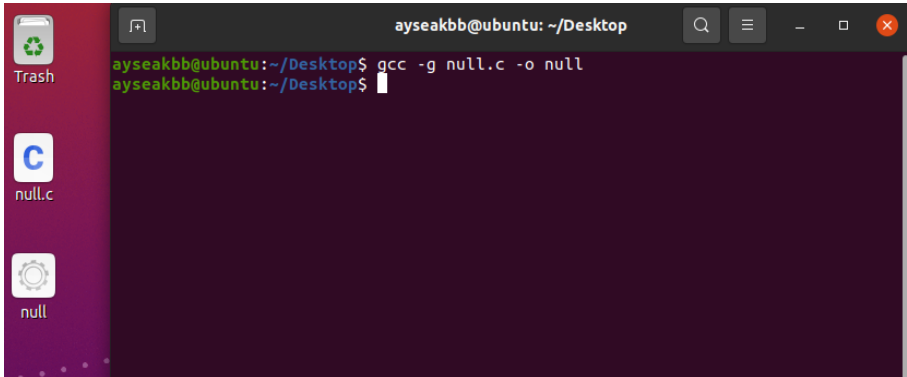


Derlenen null dosyasını çalıştırmak için `./null` kodunu kullandım ve çıktı olarak aşağıda ekran görüntüsündeki hata mesajını yazdı:

```
ayseakbb@ubuntu: ~/Desktop
ayseakbb@ubuntu:~$ cd Desktop
ayseakbb@ubuntu:~/Desktop$ gcc null.c -o null
ayseakbb@ubuntu:~/Desktop$ ./null
Segmentation fault (core dumped)
ayseakbb@ubuntu:~/Desktop$
```

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?

Bir önceki derleme işleminde hiçbir bayrak kullanmadan derleme yaptığım için önceki derleneni silip `-g` bayrağını `gcc -g null.c -o null` kodu içinde kullanarak kodu yeniden derledim.



```
ayseakbb@ubuntu: ~/Desktop
ayseakbb@ubuntu:~/Desktop$ gcc -g null.c -o null
ayseakbb@ubuntu:~/Desktop$
```

The screenshot shows a terminal window with the command `gcc -g null.c -o null` executed. On the left side of the desktop, there are icons for 'Trash', 'null.c', and 'null'.

Yazılan koddaki hatayı ayıklamak için gdb'yi (GNU Debugger) öncelikle `gdb null` diyerek hazır hale getirdim:

```
ayseakbb@ubuntu: ~/Desktop
ayseakbb@ubuntu:~/Desktop$ gcc -g null.c -o null
ayseakbb@ubuntu:~/Desktop$ gdb null
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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:
<http://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...
(gdb) 
```

Daha sonra `run` yazarak gdb'yi çalıştırdım. Yapılan hata ayıklama sonucunda: "Program received signal SIGSEGV, Segmentation fault." hatasını verdi.

```
(gdb) run
Starting program: /home/ayseakbb/Desktop/null

Program received signal SIGSEGV, Segmentation fault.
0x00005555555551b5 in main () at null.c:12
12  printf("Pointer değeri = %d", *pi); //Pointer değeri ekrana yazdırıl
ır.
(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?

Sorudaki kodu direkt kullandığımda hata aldım. Yaptığım araştırmalarda kodun aslının: `valgrind -tool=memcheck -leak-check=yes ./null` olduğunu gördüm. Bu kodu kullandığımda ise aşağıdaki çıktıyı elde ettim. Bu çıktıya göre ulaşılabilir herhangi bir adres bulunamamış. Yani `stack`'te herhangi bir adres bulunamadı. Aynı zamanda `malloc` gibi bir fonksiyon aracılığıyla bellekte de yer ayrılmamış.

Heap özetinde ise, bellekte sadece bir yer tahsisinin olduğu (i değişkeni), onun da program sonunda yerinin boşaltıldığını söylemekte. Yani değeri daha sonradan null olarak atanan pointer'ın (*pi) heap bellekte bir karşılığının olmadığını buna bağlı olarak da stack'te de bir adresin tutulmadığını göstermektedir.

```

ayseakbb@ubuntu: ~/Desktop
ayseakbb@ubuntu:~/Desktop$ valgrind --tool=memcheck --leak-check=yes ./null
==4901== Memcheck, a memory error detector
==4901== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4901== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4901== Command: ./null
==4901==
==4901== Invalid read of size 4
==4901==    at 0x1091B5: main (null.c:12)
==4901== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==4901==
==4901== Process terminating with default action of signal 11 (SIGSEGV)
==4901== Access not within mapped region at address 0x0
==4901==    at 0x1091B5: main (null.c:12)
==4901== If you believe this happened as a result of a stack
==4901== overflow in your program's main thread (unlikely but
==4901== possible), you can try to increase the size of the
==4901== main thread stack using the --main-stacksize= flag.
==4901== The main thread stack size used in this run was 8388608.
İ değeri = 5==4901==
==4901== HEAP SUMMARY:
==4901==    in use at exit: 0 bytes in 0 blocks
==4901==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==4901==
==4901== All heap blocks were freed -- no leaks are possible
==4901==
==4901== For lists of detected and suppressed errors, rerun with: -s
==4901== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Segmentation fault (core dumped)
ayseakbb@ubuntu:~/Desktop$

```

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)?

Soruda verilen isterlere göre malloc() fonksiyonu aracılığıyla daha önceden null değeri olan pointer'a bellekte bir yer tahsisi yapıldı ancak free() fonksiyonu kullanılmadığı için program bittikten sonra o yerin boşaltılması yapılmadı.

Bu programı çalıştırdığımda herhangi bir sorunla karşılaşmadım. Yine de herhangi bir sorun var mı diye kontrol amaçlı sırasıyla gdb ve valgrind kodlarını denedim.

```

Open  ▼  for forgetFree.c  Save  ≡
~/Desktop
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5
6     int *pi,i;        //pi pointer'ı ve i değişkeni tanımlandı.
7     i = 5;            //i değişkenine değer atandı
8     pi = &i;          //i'nin adresi pointer'a aktarıldı
9     pi = NULL;        //pointer'daki adresi null yaparak değişkeni referanssız hale getirildi.
10    pi = (int*)malloc(sizeof(int)); //Pointer için bellekte ayrı bir adres tahsis edilir.
11
12    printf("i değeri = %d\n", i); //i değeri ekrana yazdırılır.
13    printf("i adresi = %p\n", &i); //i adresi ekrana yazdırılır.
14    printf("Pointer değeri = %d\n", *pi); //Pointer'ın değeri ekrana yazdırılır
15    printf("Pointer adresi = %p\n", pi); //Pointer adresi ekrana yazdırılır.
16
17    return 0;
18 }

```

gdb' yi bir önceki seferde anlattığım gibi önce hazırladım sonra da çalıştırdım. Çıktı olarak çalışabilir bir program olduğunu aynı zamanda kalitesiz bir işlem olduğunu söyledi. gdb' yi kullanınca bu kalitesizliğe neyin sebep olduğunu bilmiyoruz

```

ayseakbb@ubuntu: ~/Desktop
ayseakbb@ubuntu:~/Desktop$ gcc -g null.c -o null
ayseakbb@ubuntu:~/Desktop$ gdb null
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 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:
<http://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...
(gdb) run
Starting program: /home/ayseakbb/Desktop/null
i değeri = 5
i adresi = 0x7fffffffde4c
Pointer değeri = 0
Pointer adresi = 0x5555555592a0
[Inferior 1 (process 6397) exited normally]
(gdb)

```

Aşağıdaki fotoğrafta valgrind' in kullanıldığı hali vardır. valgrind' i kullandığımda ise i değerini ve adresini yazdırırken bir sıkıntı yaşamıyor. Ancak pointer değerini yazdırırken bazı uyarılar veriyor. Sebebi bellekte pointer için bir yer ayrıldı lakin o yere herhangi bir değer atanmadı. Yani çıktı olarak ekrana ne yazdıracağını bilmiyor. Bu yüzden ekrana sıfır (0) değerini yazdırıyor.

Heap özetine bakacak olursak toplamda iki yer tahsisinin olduğunu ancak onlardan sadece birinin boşaltıldığını diğerinin ise sızıntı oluşturduğunu söylüyor. Yani yerinin serbest bırakılmadığını ifade ediyor.

En sonda ise sızıntı (kaçak) özeti var. Burada da sızıntıya sebep olan verinin yapısı ifade ediliyor. Bir bloktan oluşan ve boyutu 4 byte boyutunda bir veri kaçak yapıyor.

```

ayseakbb@ubuntu: ~/Desktop
ayseakbb@ubuntu:~/Desktop$ valgrind --tool=memcheck --leak-check=yes ./null
==6414== Memcheck, a memory error detector
==6414== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6414== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6414== Command: ./null
==6414==
i degeri = 5
i adresi = 0x1fffffd9c
==6414== Conditional jump or move depends on uninitialised value(s)
==6414== at 0x48D0958: __vfprintf_internal (vfprintf-internal.c:1687)
==6414== by 0x48BAD3E: printf (printf.c:33)
==6414== by 0x10920F: main (null.c:14)
==6414==
==6414== Use of uninitialised value of size 8
==6414== at 0x48B469B: _itoa_word (_itoa.c:179)
==6414== by 0x48D0574: __vfprintf_internal (vfprintf-internal.c:1687)
==6414== by 0x48BAD3E: printf (printf.c:33)
==6414== by 0x10920F: main (null.c:14)
==6414==
==6414== Conditional jump or move depends on uninitialised value(s)
==6414== at 0x48B46AD: _itoa_word (_itoa.c:179)
==6414== by 0x48D0574: __vfprintf_internal (vfprintf-internal.c:1687)
==6414== by 0x48BAD3E: printf (printf.c:33)
==6414== by 0x10920F: main (null.c:14)
==6414==
==6414== Conditional jump or move depends on uninitialised value(s)
==6414== at 0x48D1228: __vfprintf_internal (vfprintf-internal.c:1687)
==6414== by 0x48BAD3E: printf (printf.c:33)
==6414== by 0x10920F: main (null.c:14)
==6414==
==6414== Conditional jump or move depends on uninitialised value(s)
==6414== at 0x48D06EE: __vfprintf_internal (vfprintf-internal.c:1687)
==6414== by 0x48BAD3E: printf (printf.c:33)
==6414== by 0x10920F: main (null.c:14)
==6414==
Pointer degeri = 0
Pointer adresi = 0x4a4e040
==6414==
==6414== HEAP SUMMARY:
==6414== in use at exit: 4 bytes in 1 blocks
==6414== total heap usage: 2 allocs, 1 frees, 1,028 bytes allocated
==6414==
==6414== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6414== at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==6414== by 0x1091C4: main (null.c:10)
==6414==
==6414== LEAK SUMMARY:
==6414== definitely lost: 4 bytes in 1 blocks
==6414== indirectly lost: 0 bytes in 0 blocks
==6414== possibly lost: 0 bytes in 0 blocks
==6414== still reachable: 0 bytes in 0 blocks
==6414== suppressed: 0 bytes in 0 blocks
==6414==
==6414== Use --track-origins=yes to see where uninitialised values come from
==6414== For lists of detected and suppressed errors, rerun with: -s
==6414== ERROR SUMMARY: 6 errors from 6 contexts (suppressed: 0 from 0)
ayseakbb@ubuntu:~/Desktop$

```

Yukarıdaki açıklamalarıma göre valgrind' i kullanmamız, hatanın tespitinde ve kolaylıkla çözülmesinde daha etkilidir.

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?

Bizden istenene göre aşağıdaki kodu oluşturdum ve gcc -g data.c -o data kodunu ilk soruda olduğu gibi kullanarak kodu derledim. Kod artık çalıştırılabilir bir formattadır.

```

data.c
~/Desktop
Save

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5
6     int* data; //data isminde dinamik bellek yapısına uygun ptr tanımladım.
7     data = (int *)malloc(25 * sizeof(int)); //malloc kullanarak bellekte yer ayırdım.
8
9     data[100] = 0; //bellekte yeri olmayan 100. indexe 0 değerini atadım.
10    printf("Deger: %d\n", data[100]); //data [100] değerini ekrana yazdırmaya çalıştım.
11
12    free(data); //bellekte dizi için ayırdığım yerleri tekrardan serbest bıraktım.
13    return 0;
14 }

```

Kodu direkt terminalden çalıştırdığım zaman aşağıdaki ekran görüntüsünde de görüldüğü gibi herhangi bir hata almıyorum ve sorunsuz bir şekilde çalışıyor.

```

ayseakbb@ubuntu: ~/Desktop
data.c
data

ayseakbb@ubuntu:~/Desktop$ gcc -g data.c -o data
ayseakbb@ubuntu:~/Desktop$ ./data
Deger: 0
ayseakbb@ubuntu:~/Desktop$

```

valgrind' i kullandığım zaman (valgrind --tool=memcheck --leak-check=yes ./data) ise aşağıda görüldüğü gibi iki tane hatanın olduğundan bahsediyor. İlk olarak 4 boyutluk (bir int'lik) yazma hatası olduğundan bahsediyor. Tüm bellek boyunca böyle bir değişken için yer ayırımı yapılmamış hatası veriyor. İkinci olarak da okuma hatasının olduğunu söylüyor. Yazma hatasındakine benzer şekilde okunması istenen verinin bellekte herhangi bir karşılığının olmadığını ve bunun da durumu geçersiz kıldığını söylüyor.,


```

ayseakbb@ubuntu:~/Desktop$ valgrind --tool=memcheck --leak-check=yes ./data
==4220== Memcheck, a memory error detector
==4220== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4220== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==4220== Command: ./data
==4220==
==4220== Invalid write of size 4
==4220==    at 0x1091AD: main (data.c:9)
==4220==    Address 0x4a4e1d0 is 224 bytes inside an unallocated block of size 4,194,032 in arena "
client"
==4220==
==4220== Invalid read of size 4
==4220==    at 0x1091BD: main (data.c:10)
==4220==    Address 0x4a4e1d0 is 224 bytes inside an unallocated block of size 4,194,032 in arena "
client"
==4220==
Deger: 0
==4220==
==4220== HEAP SUMMARY:
==4220==    in use at exit: 0 bytes in 0 blocks
==4220==    total heap usage: 2 allocs, 2 frees, 1,124 bytes allocated
==4220==
==4220== All heap blocks were freed -- no leaks are possible
==4220==
==4220== For lists of detected and suppressed errors, rerun with: -s
==4220== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
ayseakbb@ubuntu:~/Desktop$

```

Bu program aslında çalışabilir ancak tam manasıyla doğru değildir.

6. Bir tamsayılar dizisine 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?

```

Open  ▾  *dealloc_then_print.c  Save  -  □  ×
~/Desktop
1#include <stdio.h>
2#include <stdlib.h>
3
4int main(void){
5
6    int* numbers; //data isminde dinamik bellek yapısına uygun ptr tanımladım.
7    numbers = (int *)malloc(5* sizeof(int)); //malloc kullanarak bellekte 20 byte'lık yer ayırdım.
8
9    free(numbers); //bellekte dizi için ayırdığım yerleri tekrardan serbest bıraktım.
10
11    printf("%d\n",numbers[0]); //diziden bir veriyi yazdırmaya çalıştım.
12    return 0;
13}

```

İstenilenlere göre dosya adı dealloc_then_print.c olan bir kod yazdım. Bu kod numbers adında tam sayılardan oluşan 20 byte'lık dinamik bir dizi oluşturdum sonrasında da bellekte ayırdığım yeri free kullanarak serbest bıraktım. Serbest bıraktıktan sonra dizinin sıfıncı indeksinde bulunan değeri yazdırmaya çalıştım. Çıktı olarak ise:

```

ayseakbb@ubuntu:~/Desktop$ gcc -g dealloc_then_print.c -o dealloc_then_print
ayseakbb@ubuntu:~/Desktop$ ./dealloc_then_print
0
ayseakbb@ubuntu:~/Desktop$

```

Yukarıdaki ekran görüntüsünü aldım. Yani herhangi bir sorun ile karşılaşmadım.

```

ayseakbb@ubuntu: ~/Desktop
ayseakbb@ubuntu:~/Desktop$ valgrind --tool=memcheck --leak-check=yes ./dealloc_then_print
==5326== Memcheck, a memory error detector
==5326== Copyright (c) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5326== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==5326== Command: ./dealloc_then_print
==5326==
==5326== Invalid read of size 4
==5326==    at 0x1091B3: main (dealloc_then_print.c:11)
==5326== Address 0x4a4e040 is 0 bytes inside a block of size 20 free'd
==5326==    at 0x483CA3F: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==5326==    by 0x1091AE: main (dealloc_then_print.c:9)
==5326== Block was alloc'd at
==5326==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==5326==    by 0x10919E: main (dealloc_then_print.c:7)
==5326==
0
==5326==
==5326== HEAP SUMMARY:
==5326==   in use at exit: 0 bytes in 0 blocks
==5326== total heap usage: 2 allocs, 2 frees, 1,044 bytes allocated
==5326==
==5326== All heap blocks were freed -- no leaks are possible
==5326==
==5326== For lists of detected and suppressed errors, rerun with: -s
==5326== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
ayseakbb@ubuntu:~/Desktop$

```

valgrind'i kullandığımda ise ilk başta bellekte malloc kullanarak yerin tahsis edildiğinden bahsediyor (Block was alloc' d at). Bu belleği free kullanarak boşalttıktan sonra dizinin sıfırıncı indeksindeki değeri yazdırmaya çalıştığım için invalid read of size 4 hatası veriyor. Bu hataya göre bellekte 20 byte'lık alan serbest bırakıldığı için öyle bir veriye ulaşamıyor.

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?

```

Open  funny_free.c  Save
~/Desktop
1#include <stdio.h>
2#include <stdlib.h>
3
4int main(void){
5
6    int* numbers; //data isminde dinamik bellek yapısına uygun ptr tanımladım.
7    numbers = (int *)malloc(5* sizeof(int)); //malloc kullanarak bellekte 20 byte'lık yer ayırdım.
8
9    free(&numbers[10]); //Free'ye olması gerekenden farklı bir değer atadım.
10
11    return 0;
12}

```

Yukarıdaki gibi 20 boyutlu dinamik bir bellek tanımladım. Ancak bellekte yer tahsis ettiğim alanları free ile boşaltmam gerekirken ben free'ye hatalı bir değer girdim.

```

ayseakbb@ubuntu:~/Desktop$ gcc -g funny_free.c -o funny_free
ayseakbb@ubuntu:~/Desktop$ ./funny_free
free(): invalid pointer
Aborted (core dumped)
ayseakbb@ubuntu:~/Desktop$

```

Bu kodu derleyip çalıştırdığımda ise: “free(): invalid pointer , Aborted (core dumped)” hatası ile karşılaştım. Bu hatanın ne olduğunu daha iyi anlamak için

valgrind özelliğini -s bayrağı ile birlikte kullandım ve çıktı olarak aşağıdaki ekran görüntüsünü elde ettim:

```

ayseakbb@ubuntu: ~/Desktop
ayseakbb@ubuntu:~/Desktop$ valgrind --tool=memcheck --leak-check=yes -s ./funny_free
==5789== Memcheck, a memory error detector
==5789== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5789== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==5789== Command: ./funny_free
==5789==
==5789== Invalid free() / delete / delete[] / realloc()
==5789==    at 0x483CA3F: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==5789==    by 0x109192: main (funny_free.c:9)
==5789== Address 0x444e068 is 20 bytes after a block of size 20 alloc'd
==5789==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==5789==    by 0x10917E: main (funny_free.c:7)
==5789==
==5789== HEAP SUMMARY:
==5789==    in use at exit: 20 bytes in 1 blocks
==5789==    total heap usage: 1 allocs, 1 frees, 20 bytes allocated
==5789==
==5789== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==5789==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==5789==    by 0x10917E: main (funny_free.c:7)
==5789==
==5789== LEAK SUMMARY:
==5789==    definitely lost: 20 bytes in 1 blocks
==5789==    indirectly lost: 0 bytes in 0 blocks
==5789==    possibly lost: 0 bytes in 0 blocks
==5789==    still reachable: 0 bytes in 0 blocks
==5789==    suppressed: 0 bytes in 0 blocks
==5789==
==5789== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
==5789==
==5789== 1 errors in context 1 of 2:
==5789== Invalid free() / delete / delete[] / realloc()
==5789==    at 0x483CA3F: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==5789==    by 0x109192: main (funny_free.c:9)
==5789== Address 0x444e068 is 20 bytes after a block of size 20 alloc'd
==5789==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==5789==    by 0x10917E: main (funny_free.c:7)
==5789==
==5789== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
ayseakbb@ubuntu:~/Desktop$

```

Yukarıdaki ekran görüntüsüne göre, geçersiz bir serbest bırakma, silme ya da yeniden yer tahsis etme işlemi olduğundan bahsediyor. Aynı zamanda bloktan oluşan ve yirmi boyutlu bir verinin de kaçak(sızıntı) yaptığını, yani yer tahsis edildi ancak yer boşaltılmadığı, anlatıyor.

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.

Yukarıdaki isterlere göre aşağıdaki kodu oluşturdum. Başlangıç değeri olarak 3 verdim. Kullanıcı üçten fazla veri girmek istediğinde `malloc` işlevi ile tahsis ettiğimiz yerin dolu olduğu ile ilgili bir uyarı aldıktan sonra `realloc` ile yer açtığı dönütünü vererek daha fazla veri almaya devam eden bir kod idir.

```

Open  realloc.c  Save
~/Desktop

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6     int size=0, capacity=0, *elements;
7     int cevap, initialSize=3;
8     elements = (int*)malloc(initialSize * sizeof(int));
9
10    if(elements == NULL){
11        printf("Bellekte malloc ile yer tahsisi yapılamadı.");
12    }else{
13        printf("malloc kullanarak bellekte yer tahsis etme başarılı. \n");
14        printf("\n element = %p c adreste\n", elements);
15        do {
16            printf("\n Bir deger giriniz: ");
17            scanf("%d", &elements[size]);
18            printf("Ekleme ister misiniz? (evet=1 / hayır=0): ");
19            scanf("%d", &cevap);
20            printf("-----\n");
21            if(size == initialSize){
22                if (cevap == 1){
23                    initialSize *= 2;
24                    elements = (int*)realloc(elements, initialSize * sizeof(int));
25                    if (elements == NULL) {
26                        printf("Bellekte realloc ile yer tahsisi yapılamadı.");
27                    }
28                    else {
29                        printf("realloc kullanarak bellekte yer tahsis etme başarılı. \n");
30                        printf("\n element = %pc adreste\n", elements);
31                    }
32                }
33            }
34            if(size != initialSize){
35                if (cevap == 1) {
36                    size++;
37                }
38            }
39            while (cevap == 1);
40            for (int i = 0; i <= size; i++) {
41                printf("Vektorun %d . elemanı: %d\n ", i+1, elements[i]);
42            }
43            free(elements);
44        }
45        return 0;
46    }

```

Performans açısından gayet iyidir. Normal dizi tanımlamalarında olduğu gibi sabit bir boyutla uğraşıp o boyut dolunca gelen hataları çözmeye uğraşmıyoruz. Eğer bellek dolarsa var olan belleğin iki katı kadar yer oluşturuyor ve sorun almadan yolumuza devam edebiliriz. Bu kodu yazarken veri yapılarında var olan “Hash Table” konusu ile olan benzerliği dikkatimi çekti ve konuyu daha rahat anlamamı sağladı.

Bağlı listeler ile `realloc` işlevini kıyasladığım zaman ikisinin de kendine göre olumlu ve olumsuz yönlerinin olduğunu söyleyebilirim. Bağlı listeler ekleme veya çıkarmanın fazlaca yapıldığı sistemlerde oldukça verimlidir ve hızlıdır. Sebebi iki bağlı liste arasındaki bağı koparıp yeni elemanı ekleyip bağlantıları kurmak; tüm listeyi silinen ya da eklenen eleman sayısı kadar aşağı ya da yukarı kaydırmaktan çok daha hızlıdır. `Realloc` fonksiyonu ise daha çok herhangi bir veriyi arama veya var olan verileri sıralama konusunda bağlı listelere kıyasla daha verimlidir ve hızlıdır. Kısacası ikisinden herhangi biri için “Bu çok iyidir, mükemmeldir” gibi bir yorum beklenemez.

Yapılacak iş için en uygun yöntem şeklinde bahsedebilirim ve bu durum da işten işe değişiklik gösterir.

`valgrind`'i kullandığımda karşıma böyle bir çıktı geldi. Bu sorunun cevapları arasındaki ilk paragrafta da bahsettiğim gibi öncelikle `malloc()` fonksiyonunu

```

ayseakbb@ubuntu: ~/Desktop
ayseakbb@ubuntu:~/Desktop$ gcc -g realloc.c -o realloc
ayseakbb@ubuntu:~/Desktop$ valgrind --tool=memcheck --leak-check=yes ./realloc
==8874== Memcheck, a memory error detector
==8874== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==8874== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==8874== Command: ./realloc
==8874==
malloc kullanarak bellekte yer tahsis etme başarılı.

element = 0x4a4e040 c adreste

Bir deger giriniz: 1
Ekleme ister misiniz? (evet=1 / hayir=0): 1
-----

Bir deger giriniz: 2
Ekleme ister misiniz? (evet=1 / hayir=0): 1
-----

Bir deger giriniz: 3
Ekleme ister misiniz? (evet=1 / hayir=0): 1
-----

Bir deger giriniz: 4
==8874== Invalid write of size 4
==8874== at 0x48C1335: __vfprintf_internal (vfprintf-internal.c:1895)
==8874== by 0x48BC161: __isoc99_scanf (isoc99_scanf.c:30)
==8874== by 0x1092C7: main (realloc.c:16)
==8874== Address 0x4a4e04c is 0 bytes after a block of size 12 alloc'd
==8874== at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-and64-linux.so)
==8874== by 0x109249: main (realloc.c:7)
==8874==
Ekleme ister misiniz? (evet=1 / hayir=0): 1
-----

realloc kullanarak bellekte yer tahsis etme başarılı.

element = 0x4a4e910c adreste

Bir deger giriniz: 5
Ekleme ister misiniz? (evet=1 / hayir=0): 0

```

kullanarak bir başlangıç değerine göre bellekte yer tahsis ettim. Daha sonra veri girişlerini aldım. Girilen veriler ayırdığım alana sığmamaya başlayınca (koddaki karşılığı => `if(size == initialSize)`) `malloc()` işlevi, bellekte ayırdığım yerin dolduğu hatasını fırlattı. Bu dönütün hemen ardından da `realloc()` fonksiyonu devreye girerek var olan başlangıç boyutunun iki katı kadar bellekte yeni bir adres tanımladı ve bir önceki adreste tuttuğum verileri tek tek yeni oluşturduğum adrese atadı. Bu sayede de yeni gelecek olan kullanıcı verileri için bellekte yer açılmış oldu.

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