

C Programlama Arayüzleri II: Bellek Yönetimi

Bariş Şimşek, <simsek ~ enderunix.org>
<http://www.enderunix.org/simsek>
Yazılım Geliştiricisi

Daha önceki “Memory Management for System Programmers” (1) makalesinde bahsedildiği gibi bazı C programları çalışma zamanında (runtime) kendisine bellek tahsis edilmesini isteyebilmektedir. Standart C kütüphanesi bunun için dört fonksiyon sunmaktadır: **malloc**, **calloc**, **realloc** ve **free**.

Ne yazık ki C programların çoğunda çıkan hatalar genelde dinamik bellek yönetimi ve işaretçiler (pointer) ile ilgili olmaktadır. İşte bu nedenle bu makalede adı geçen fonksiyonları güvenle kullanmamızı sağlayacak bir arayüz (interface) tasarlamaya çalışacağım. Bu makaleyi okumadan önce adı geçen makaleyi, özellikle “Pointers” kısmını okumanız faydalı olacaktır. Bu makale, David R. Hanson'un kitabı (2) üzerine bina edilmiştir. Hanson'un söylediklerine kendimden çok fazla bir şey eklemiş değilim.

TUZAKLAR

Aşağıdaki kod parçasına bakalım:

Kod 1:

```
char buf[256];  
  
...  
p = buf;  
...  
free(p);
```

Buraya kodun küçük bir kısmını aldığımız için hatayı görmek oldukça kolay. Ancak binlerce satırın olduğu büyük kodlarda böyle bir hata, iç içe dallanmalar nedeniyle iyi bir programcının dahi gözünden kaçabilir. O nedenle buraya yalnızca ilgili kısmını aldığım kodun binlerce satırlık bir kod içine dağıldığı varsayımı yapılsın. Böylece bu tuzağın sanıldığı kadar basit olmadığını göreceksiniz.

Kod 1, tahsis edilmemiş bir bellek alanının işletim sistemine geri verilmeye çalışılması nedeni ile hatalıdır.

Kod 2:

```
buf = malloc(256);  
...  
free(buf);  
...  
free(buf);
```

Kod 2, serbest bırakılmış bir bellek bölgesinin tekrar serbest bırakılmaya çalışılması nedeni ile hatalıdır.

Diğer makalede de verilen aşağıdaki kod basit bir “bellek sızıntısı” (memory leak) hatasını göstermektedir. Burada programcı koda bakarak hem tmp hem de str'yi işletim sistemine verdiğini zannetmektedir. Ancak arada bir yerde tmp = str ataması yapıldığından gerçekte tmp serbest bırakılmamıştır.

Kod 3:

```
char* str;
char* tmp;

str = malloc(sizeof(char)*32);
tmp = malloc(sizeof(char)*32);
fscanf(stdin, "%s", str);
tmp = str;

do_something_with_tmp();
do_something_with_str();
free(str);
free(tmp);

return 0;
```

Arayüz

Tanımlanacak arayüz ile bellek yönetim rutinlerini en güvenli şekilde kullanmayı amaçlamaktayız. Bu makalede yalnız alloc() rutini için bir arayüz tasarlayacağım.

Programcıya alloc() yerine NEW() makrosunu sunacağız:

```
#define NEW(p) ( (p) = ALLOC((long) sizeof *(p)) )
```

Bu makro kendisi ne kadar bir boyutta bellek ayırması gerektiğini hesap edip ALLOC() isimli bir başka makromuzu çağırıyor. Programcı ayrılacak bellek boyutu bilgisinden yalıtılıyor. Bellek boyutu herhangi bir struct ile çalışacak şekilde hesap edilmektedir.

```
#define ALLOC(nbytes) mem_alloc((nbytes), __FILE__, __LINE__)

extern void mem_alloc(long nbytes, const char *file, int line);
```

Bu makrodaki ilk parametre 'nbytes' olup alloc() rutini ile ayıracağımız bellek alanının boyutunu belirler. İkinci ve üçüncü parametreler ise yalnızca hata ayıklamada (debug) yardımcı olması içindir. __FILE__ tanımlayıcısı derleme zamanında kaynak kod dosyasının ismi ile, __LINE__ tanımlayıcısı ise makronun çağırıldığı satır numarası ile değiştirilir.

NEW makrosu programcı önünde iken (front-end), ALLOC() arka planda (back-end) kalacaktır. O nedenle NEW makrosunu olabildiğince sade, ALLOC makrosunu ise olabildiğince fonksiyonel tanımlamaya çalışıyoruz.

Eğer mem_alloc istenen belleği ayıramaz ise mem_error() rutinini çağırarak ve hata ayıklaması için __FILE__ ve __LINE__ değerlerini bu rutine parametre olarak verecektir.

Arayüzü bu şekilde tanımlamış olduk. Şimdi arkada işleri yapacak olan `mem_alloc` rutinini yazmaya çalışalım. Ancak öncelikle C kütüphanesinin sağladığı `malloc()` rutinini inceleyelim. Çünkü arayüz dediğimiz makro aslında C kütüphanesinin sağladığı rutin sarmalanmış (encapsulation) daha güvenli bir paket haline getirilmiş şeklidir.

```
void malloc(size_t size);
```

`size_t` olabilecek en büyük nesne büyüklüğüdür ve standart C kütüphanesinde büyüklükleri göstermek için kullanılır. Pratikte işaretli tamsayıdır (unsigned integer veya long).

```
void mem_alloc(long nbytes, const char *file, int line)
{
    void *ptr;

    assert(nbytes > 0);

    ptr = malloc(nbytes);
    if (ptr == NULL)
        mem_error(file, line);

    return ptr;
}
```

`nbytes` negatif bir değer olsa bile çoğu platformda `size_t`'nin işaretli (unsigned) olarak tanımlanmasından dolayı bir hataya neden olmaz. Örneğin `malloc(-1)`'in hata vereceği açıktır. Ancak çoğu sistemde `-1` işaretli tamsayıya çevrileceğinden (ki bu çok büyük bir pozitif sayıdır) hata vermeyecektir. Ancak biz yine de garantiye almak için `assert` ile ifade (`nbytes > 0`) kontrolü yapıyoruz.

`mem_error()`, hata kontrolü için kullanılan fonksiyondur. Bu herhangi bir fonksiyon olabilir. Bu fonksiyonu da bir sonraki makalede tanımlayacağım.

Daha önceki makaleden hatırlayacağınız gibi C dili, çöp toplama (garbage collection) yapmamakta idi. Bu nedenle ayrılan bellek alanları işletim sistemine geri verilmez ise “bellek sızması” oluşur. Bir bellek alanının serbest bırakılması için C kütüphanesi `free()` rutinini sunmaktadır. Bu rutin için de bir arayüz yazacağız.

```
#define FREE(ptr) ((void) (mem_free((ptr), __FILE__, __LINE__), (ptr) = 0))

extern void mem_free(void *ptr, const char *file, int line);
```

Programcıya sunulan `FREE` makrosu parametre olarak iade edilmek istenen bölgeyi gösteren işaretçiyi alır. Bu makro arka planda `free()` rutininden daha fonksiyonel olan `mem_free()` rutinini çağırır ve ardından işaretçiyi `NULL` olacak şekilde atar. Böylece serbest bırakılan işaretçinin başıboş olarak rastgele bir bellek bölgesini göstermesinin önüne geçilir. `mem_free()`, işaretçi `NULL` olur ise herhangi bir şey yapmaz. `NULL` değil ise `free()` rutinini kullanarak işaretçiyi serbest bırakır.

```
void mem_free(void *ptr, const char *file, int line) {
    if (ptr)
        free(ptr);
}
```

Standart `free()` parametre olarak `NULL` işaretçi alabilir. Ancak bazı eski C kütüphaneleri bunu kabul etmeyebilir.

Kolay anlaşılabilmesi için arayüzler basit olarak tasarlanmıştır. Bu arayüzlere ilave kontroller eklemek mümkündür. Ayrıca `mem_error()` rutini bu belgede açıklanmamıştır. Başka bir makalede hata yönetimi için arayüz tasarlama üzerinde durulacaktır.

Belge Hakkında

Bu belge kaynak gösterilmek şartı ile kullanılabilir.

Bu belge “GNU Free Documentation License” altında yayınlanabilir, dağıtılabilir.

© Copyleft Barış Şimşek

```
Copyright (c) Baris Simsek.  
Permission is granted to copy, distribute and/or modify this document  
under the terms of the GNU Free Documentation License, Version 1.2  
or any later version published by the Free Software Foundation;  
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.  
A copy of the license is included in the section entitled "GNU  
Free Documentation License".
```

- (1) <http://www.enderunix.org/simsek/articles/memory.pdf>, Baris Simsek
- (2) C Interfaces and Implementations, Addison-Wesley, David R. Hanson