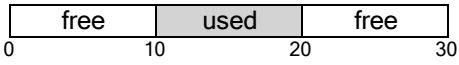
17

Free-Space Management

Bu bölümde, herhangi bir bellek yönetim sisteminin temel bir yönünü tartışmak için belleği sanallaştırma tartışmamızdan küçük bir sapma yapacağız, bu bir malloc kitaplığı (bir işlemin yığınının sayfalarını yönetme) veya işletim sisteminin kendisi olsun. (bir işlemin adres alanının bölümlerini yönetme). Spesifik olarak, **boş alan yönetimini(free-space management)** çevreleyen sorunları tartışacağız.

Sorunu daha spesifik hale getirelim. **Sayfalama(paging)** kavramını tartışırken göreceğimiz gibi, boş alanı yönetmek kesinlikle kolay olabilir. Yönettiğiniz alan sabit boyutlu birimlere bölündüğünde kolaydır; böyle bir durumda, bu sabit boyutlu birimlerin bir listesini tutmanız yeterlidir; bir müşteri bunlardan birini istediğinde, ilk girişi döndürür.

Boş alan yönetiminin daha zor (ve ilginç) hale geldiği yer, yönettiğiniz boş alanın değişken boyutlu birimlerden oluşmasıdır; bu, kullanıcı düzeyinde bir bellek ayırma kitaplığında (malloc() ve free()'de olduğu gibi) ve sanal belleği uygulamak için **bölümlemeyi(segmentation)** kullanırken fiziksel belleği yöneten bir işletim sisteminde ortaya çıkar. Her iki durumda da, mevcut sorun dış parçalanma**(external fragmentation**) olarak bilinir: boş alan farklı boyutlarda küçük parçalara bölünür ve böylece parçalanır; toplam boş alan miktarı isteğin boyutunu aşsa bile, isteği karşılayabilecek tek bir bitişik alan olmadığından sonraki istekler başarısız olabilir.



Şekil bu sorunun bir örneğini göstermektedir. Bu durumda, kullanılabilir toplam boş alan 20 bayttır; ne yazık ki, her biri 10 büyüklüğünde iki parçaya bölünmüştür. Sonuç olarak, 20 bayt boş olsa bile 15 baytlık bir istek başarısız olur. Böylece bu bölümde ele alınan soruna ulaşmış oluyoruz.

CRUX: BOŞ ALAN NASIL YÖNETİLİR

Değişken boyutlu talepler karşılanırken boş alan nasıl yönetilmelidir? Parçalanmayı en aza indirmek için hangi stratejiler kullanılabilir?

17.1 Varsayımlar

Bu tartışmanın çoğu, kullanıcı düzeyinde bellek ayırma kitaplıklarında bulunan ayırıcıların büyük geçmişine odaklanacaktır. Wilson'ın mükemmel araştırmasından [W+95] yararlanıyoruz, ancak ilgili okuyucuları daha fazla ayrıntı için kaynak belgenin kendisine gitmeye teşvik ediyoruz1.

malloc() ve free() tarafından sağlanan gibi temel bir arabirim varsayıyoruz. Spesifik olarak, void *malloc(size t size), uygulama tarafından istenen bayt sayısı olan tek bir parametre olan size alır; o boyuttaki (veya daha büyük) bir bölgeye bir işaretçiyi (belirli bir türden veya C lingo'da **geçersiz bir işaretçi(void pointer**)) geri verir. Tamamlayıcı rutin void free(void *ptr) bir işaretçi alır ve karşılık gelen yığını serbest bırakır. Arayüzün ne anlama geldiğine dikkat edin: kullanıcı, alanı boşaltırken kütüphaneye boyutunu bildirmez; bu nedenle, kitaplık, yalnızca bir işaretçi verildiğinde bir bellek yığınının ne kadar büyük olduğunu anlayabilmelidir. Bunu nasıl yapacağımızı bu bölümde biraz sonra tartışacağız.

Bu kitaplığın yönettiği alan tarihsel olarak yığın olarak bilinir ve yığındaki boş alanı yönetmek için kullanılan genel veri yapısı bir tür boş listedir**(free list).** Bu yapı, yönetilen bellek bölgesindeki tüm boş alan parçalarına başvurular içerir. Tabii ki, bu veri yapısının kendi başına bir liste olması gerekmez, sadece boş alanı izlemek için bir tür veri yapısı olması gerekir.

Ayrıca, yukarıda açıklandığı gibi öncelikle dış parçalanma**(external fragmentation)** ile ilgilendiğimizi varsayıyoruz. Ayırıcılar elbette **dahili parçalanma(internal fragmentation)** sorununa da sahip olabilir; Bir tahsisatçı talep edilenden daha büyük bellek parçalarını dağıtırsa, böyle bir yığında talep edilmeyen (ve dolayısıyla kullanılmayan) herhangi bir alan dahili parçalanma olarak kabul edilir (çünkü atık tahsis edilen birim içinde meydana gelir) ve alan israfının başka bir örneğidir. Bununla birlikte, basitlik adına ve bu iki parçalanma türünden daha ilginç olduğu için, çoğunlukla dış parçalanmaya odaklanacağız.

Ayrıca, bir istemciye bellek dağıtıldığında, bellekte başka bir konuma taşınamayacağını da varsayacağız. Örneğin, bir program malloc()'u çağırırsa ve öbek içindeki bir boşluğa bir işaretçi verilirse, bu bellek bölgesi esasen program tarafından "sahiptir" (ve kitaplık tarafından taşınamaz), program onu bir karşılık yoluyla döndürene kadar. - free()'ye çağrı yapmak. Böylece, boş alanın **sıkıştırılması(compact**) mümkün değildir, bu da

[1]It is nearly 80 pages long; thus, you really have to be interested!

parçalanmayla mücadele etmek için yararlı olacaktır2. Ancak sıkıştırma, işletim sisteminde **segmentasyon(segmentation**) uygulanırken parçalanma ile başa çıkmak için kullanılabilir (segmentasyonla ilgili bahsedilen bölümde tartışıldığı gibi).

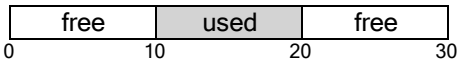
Son olarak, ayırıcının bitişik bir bayt bölgesini yönettiğini varsayacağız. Bazı durumlarda, bir tahsisatçı o bölgenin büyümesini isteyebilir; örneğin, kullanıcı düzeyinde bir bellek ayırma kitaplığı, boş alan bittiğinde yığını büyütmek için (sbrk gibi bir sistem çağrısı yoluyla) çekirdeğe çağrı yapabilir. Bununla birlikte, basitlik için, bölgenin ömrü boyunca tek bir sabit boyut olduğunu varsayacağız.

17.2 Low-level Mechanisms

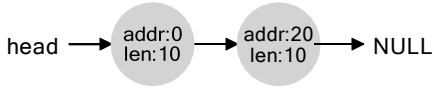
Bazı politika ayrıntılarını incelemeden önce, çoğu paylaştırıcıda kullanılan bazı ortak mekanizmaları ele alacağız. İlk olarak, herhangi bir tahsisatçının çoğunda yaygın olan teknikler olan bölme ve birleştirmenin temellerini tartışacağız. İkinci olarak, tahsis edilen bölgelerin boyutunun nasıl hızlı ve görece kolaylıkla takip edilebileceğini göstereceğiz. Son olarak, neyin ücretsiz olup neyin olmadığını takip etmek için boş alan içinde basit bir listenin nasıl oluşturulacağını tartışacağız.

Bölme ve Birleştirme

Boş bir liste, yığında hala kalan boş alanı tanımlayan bir dizi öğe içerir. Bu nedenle, aşağıdaki 30 baytlık yığını varsayalım:



Bu yığın için boş listede iki öğe bulunur. Bir giriş, ilk 10 baytlık boş segmenti (bayt 0-9) tanımlar ve bir giriş, diğer boş segmenti (bayt 20-29) tanımlar:

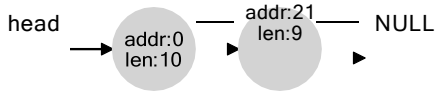


Yukarıda açıklandığı gibi, 10 bayttan büyük bir istek başarısız olur (NULL döndürülür); bu boyutta tek bir bitişik bellek parçası yok. Tam olarak bu boyut (10 bayt) için bir istek, ücretsiz parçalardan herhangi biri tarafından kolayca karşılanabilir. Ancak istek 10 bayttan küçük bir şey içinse ne olur?

Tek bir baytlık bellek talebimiz olduğunu varsayalım. Bu durumda, ayırıcı **bölme(splitting)** olarak bilinen bir eylemi gerçekleştirecektir:

[2]Once you hand a pointer to a chunk of memory to a C program, it is generally difficult to determine all references (pointers) to that region, which may be stored in other variables or even in registers at a given point in execution. This may not be the case in more strongly-typed, garbage-collected languages, which would thus enable compaction as a technique to combat fragmentation.

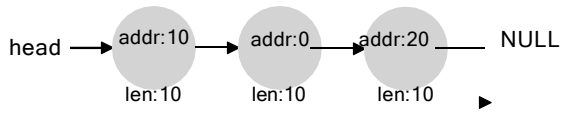
isteği karşılayabilecek ve ikiye bölebilecek boş bir bellek yığını. Arayana geri döneceği ilk parça; ikinci parça listede kalacaktır. Bu nedenle, yukarıdaki örneğimizde, 1 baytlık bir istekte bulunulursa ve ayırıcı, isteği karşılamak için listedeki iki öğeden ikincisini kullanmaya karar verirse, malloc() öğesine yapılan çağrı 20 (adresi) döndürür. 1 baytlık ayrılmış bölge) ve liste şöyle görünürdü:



Resimde, listenin temelde bozulmadan kaldığını görebilirsiniz; tek değişiklik, serbest bölgenin artık 20 yerine 21'de başlaması ve bu serbest bölgenin uzunluğunun artık sadece 93 olmasıdır. Bu nedenle, ayırma, istekler herhangi bir belirli serbest yığının boyutundan daha küçük olduğunda ayırıcılarda yaygın olarak kullanılır.

Birçok paylaştırıcıda bulunan bir sonuç mekanizması, boş alanın **birleşmesi(coalescing)** olarak bilinir. Örneğimizi bir kez daha yukarıdan alın (boş 10 bayt, kullanılmış 10 bayt ve başka bir boş 10 bayt).

Bu (küçük) yığın göz önüne alındığında, bir uygulama free(10) çağırdığında ve böylece yığının ortasındaki alanı döndürdüğünde ne olur? Bu boş alanı çok fazla düşünmeden listemize geri eklersek, şuna benzeyen bir listeyle sonuçlanabiliriz.:



Soruna dikkat edin: tüm yığın artık boş olsa da, görünüşte her biri 10 baytlık üç parçaya bölünmüştür. Bu nedenle, bir kullanıcı 20 bayt isterse, basit bir liste geçişi böyle boş bir yığın bulamaz ve hata döndürür.

Ayırıcıların bu sorunu önlemek için yaptığı şey, bir bellek parçası serbest bırakıldığında boş alanı birleştirmek. Fikir basit: bellekte boş bir yığın döndürürken, döndürdüğünüz yığının adreslerine ve yakındaki boş alan yığınlarına dikkatlice bakın; yeni serbest bırakılan alan bir (veya bu örnekte olduğu gibi iki) mevcut boş yığının hemen yanındaysa, bunları daha büyük tek bir boş yığın halinde birleştirin. Böylece, birleştirme ile son listemiz şöyle görünmelidir.:

Aslında, herhangi bir tahsis yapılmadan önce yığın listesi ilk bakışta böyle görünüyordu. Birleştirme ile, bir tahsisatçı, uygulama için büyük boş uzantıların mevcut olmasını daha iyi sağlayabilir.

[3]This discussion assumes that there are no headers, an unrealistic but simplifying assump-

® 2008–19, ARPACI-DUSSEAU

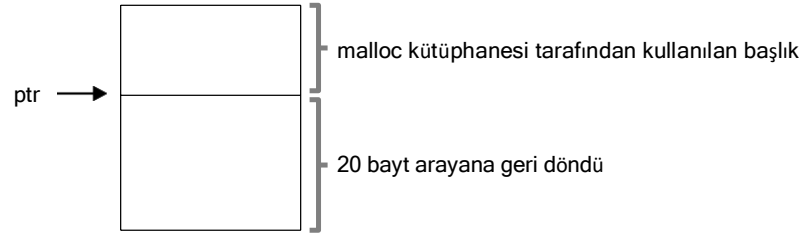


Figure 17.1: **Tahsis Edilmiş Bölge Artı Başlığı**

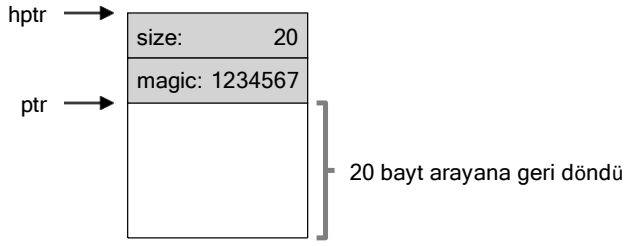


Figure 17.2: **Başlığın Özel İçeriği**

Tahsis Edilen Bölgelerin Büyüklüğünü İzleme

free(void *ptr) arabiriminin size parametresi almadığını fark etmiş olabilirsiniz; bu nedenle, bir işaretçi verildiğinde, malloc kitaplığının serbest bırakılan bellek bölgesinin boyutunu hızlı bir şekilde belirleyebileceği ve böylece alanı boş listeye geri ekleyebileceği varsayılır.

Bu görevi gerçekleştirmek için, çoğu ayırıcı, genellikle dağıtılan bellek yığınından hemen önce, bellekte tutulan bir **başlık bloğunda(header**) biraz fazladan bilgi depolar. Tekrar bir örneğe bakalım (Şekil 17.1). Bu örnekte, ptr ile gösterilen 20 bayt boyutunda ayrılmış bir bloğu inceliyoruz; malloc() adlı kullanıcıyı ve sonuçları ptr'de sakladığını hayal edin, örn., ptr = malloc(20);.

Başlık, ayrılan bölgenin boyutunu minimum düzeyde içerir (bu durumda, 20); ayrıca, ayırma işlemini hızlandırmak için ek işaretçiler, ek bütünlük denetimi sağlamak için sihirli bir sayı ve diğer bilgileri içerebilir. Bölgenin boyutunu ve sihirli bir sayıyı içeren basit bir başlık varsayalım, şöyle:

```
typedef struct {
    int size;
    int magic;
} header_t;
```

Yukarıdaki örnek, Şekil 17.2'de gördüğünüz gibi görünecektir. Kullanıcı free(ptr) öğesini çağırdığında, kitaplık başlığın nerede başladığını bulmak için basit işaretçi aritmetiğini kullanır:

```
void free(void *ptr) {
    header_t *hptr = (header_t *) ptr - 1;
    ...
```

Başlığa böyle bir işaretçi aldıktan sonra, kitaplık, sihirli sayının bir akıl sağlığı kontrolü olarak beklenen değerle eşleşip eşleşmediğini kolayca belirleyebilir (assert(hptr->magic == 1234567) ve toplam boyutunu hesaplayabilir. basit matematik yoluyla yeni serbest bırakılan bölge (yani, başlığın boyutunu bölgenin boyutuna ekleyerek). Son cümledeki küçük ama kritik ayrıntıya dikkat edin: boş bölgenin boyutu, başlığın boyutu ile kullanıcıya ayrılan alanın boyutudur. Bu nedenle, bir kullanıcı N bayt bellek istediğinde, kitaplık N boyutunda boş bir yığın aramaz; bunun yerine, N boyutunda serbest bir yığın artı başlık boyutunu arar.

Embedding A Free List

Şimdiye kadar basit serbest listemizi kavramsal bir varlık olarak ele aldık; Bu sadece yığındaki boş bellek parçalarını tanımlayan bir listedir. Fakat boş alanın içinde böyle bir listeyi nasıl oluşturabiliriz?

Daha tipik bir listede, yeni bir düğüm tahsis ederken, düğüm için alana ihtiyacınız olduğunda yalnızca malloc() öğesini çağırırsınız. Ne yazık ki, bellek ayırma kitaplığı içinde bunu yapamazsınız! Bunun yerine, listeyi boş alanın içinde oluşturmanız gerekir. Kulağa biraz garip geliyorsa endişelenmeyin; öyle, ama yapamayacağın kadar garip değil!!

Yönetmek için 4096 baytlık bir bellek parçamız olduğunu varsayalım (öbek 4kb'dir). Bunu boş bir liste olarak yönetmek için önce söz konusu listeyi başlatmamız gerekir; Başlangıçta listenin 4096 boyutunda (eksi başlık boyutu) bir girişi olmalıdır. İşte listenin bir düğümünün açıklaması:

```
typedef struct __node_t {
    int              size;
    struct __node_t *next;
} node_t;
```

Şimdi yığını başlatan ve boş listenin ilk öğesini bu boşluğa yerleştiren bazı kodlara bakalım. Yığının, sistem çağrısı mmap () çağrısı yoluyla edinilen bir miktar boş alan içinde oluşturulduğunu varsayıyoruz; Böyle bir yığın oluşturmanın tek yolu bu değil, bu örnekte bize iyi hizmet ediyor. İşte kod:

```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size   = 4096 - sizeof(node_t);
head->next   = NULL;
```

® 2008–19, ARPACI-DUSSEAU

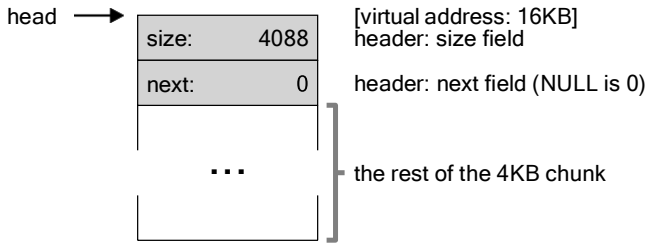


Figure 17.3: **A Heap With One Free Chunk**

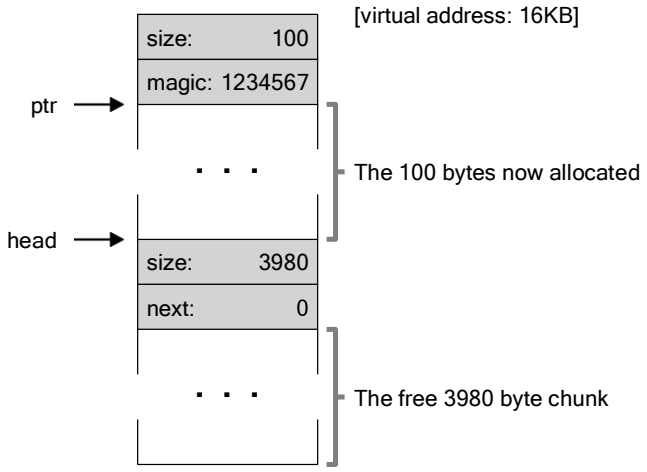


Figure 17.4: **Bir Yığın: Bir Tahsisten Sonra**

Bu kodu çalıştırdıktan sonra listenin durumu, 4088 boyutunda tek bir girişi olmasıdır. Evet, bu küçük bir yığın, ama burada bizim için güzel bir örnek teşkil ediyor. Head işaretçisi bu aralığın başlangıç adresini içerir; 16 KB olduğunu varsayalım (herhangi bir sanal adres iyi olsa da). Görsel olarak yığın, Şekil 17.3'te gördüğünüze benziyor.

Şimdi, 100 bayt boyutunda bir bellek parçasının istendiğini hayal edelim. Bu talebe hizmet vermek için, kütüphane önce talebi karşılayacak kadar büyük bir yığın bulacaktır; yalnızca bir boş yığın olduğundan (boyut: 4088), bu yığın seçilecektir. Ardından, yığın ikiye **bölünecektir(split):** talebe hizmet edecek kadar büyük bir yığın (ve yukarıda açıklandığı gibi başlık) ve kalan boş yığın. 8 Baytlık bir başlık (bir tamsayı boyutu ve bir tamsayı sihirli sayı) varsayarsak, yığındaki boşluk şimdi Şekil 17.4'te gördüğünüze benziyor.

Böylece, 100 bayt talebi üzerine kütüphane 108 bayt ayırdı

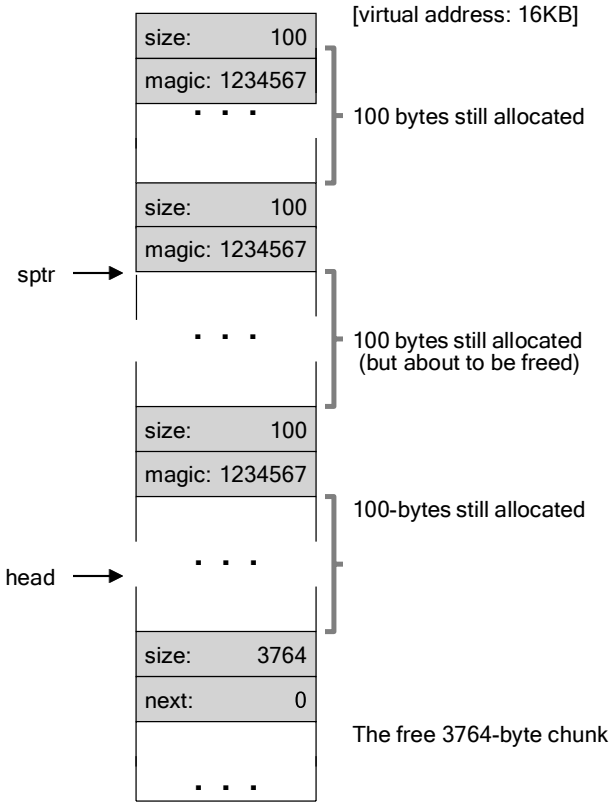


Figure 17.5: **Üç Parça Ayrılmış Boş Alan**

varolan bir boş öbek dışında bir işaretçi döndürür (yukarıdaki şekilde ptr olarak işaretlenmiştir), başlık bilgilerini daha sonra kullanmak üzere ayrılan alandan hemen önce saklar. boş () ve listedeki bir boş düğümü 3980 bayta küçültür (4088 eksi 108).

Şimdi, her biri 100 bayt (veya başlık dahil 108) olmak üzere üç ayrılmış bölge olduğunda yığına bakalım. Bu yığının bir görselleştirmesi Şekil 17.5'te gösterilmiştir.

Burada gördüğünüz gibi, yığının ilk 324 baytı artık ayrılmıştır ve bu nedenle o alanda üç başlığın yanı sıra çağıran program tarafından kullanılan üç 100 baytlık bölge görüyoruz. Ücretsiz liste ilgisiz kalıyor: sadece tek bir düğüm (baş tarafından işaret edildi), ancak şimdi sadece 3764

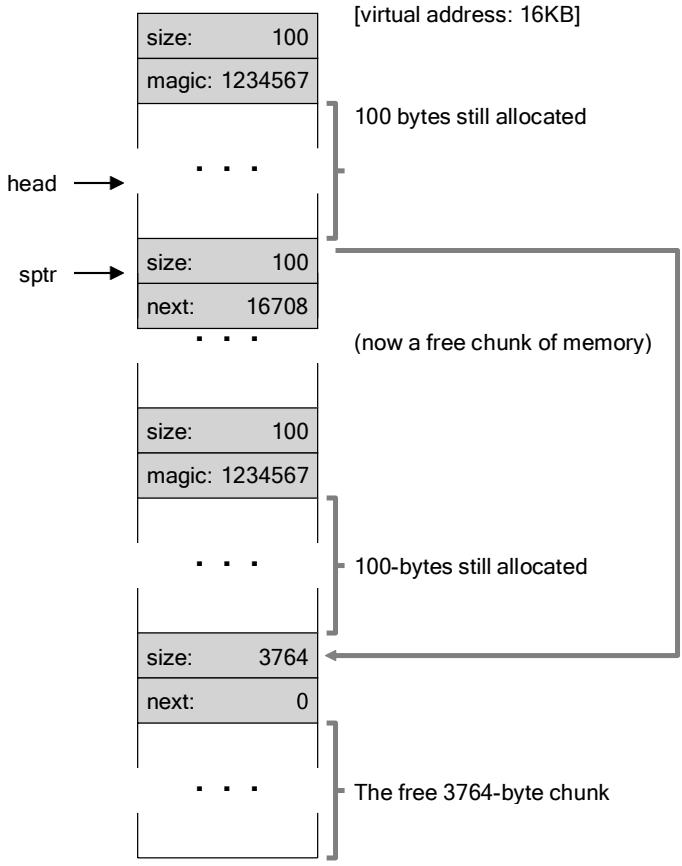


Figure 17.6: **İki parça Ayrılmış Boş Alan**

üç bölmeden sonra bayt boyutu. Ancak, çağıran program free() aracılığıyla bir miktar bellek döndürdüğünde ne olur?

Bu örnekte, uygulama boş (16500) çağırarak ayrılan belleğin orta yığınını döndürür (16500 değeri, bellek bölgesinin başlangıcı olan 16384'ü önceki yığının 108'ine ve bu yığın için üstbilginin 8 baytına ekleyerek elde edilir). Bu değer önceki şemada sptr işaretçisi ile gösterilmiştir.

Kütüphane hemen serbest bölgenin boyutunu bulur ve ardından serbest parçayı tekrar serbest listeye ekler. Boş listenin başına eklediğimizi varsayarsak, boşluk şimdi şöyle görünür (Şekil 17.6).

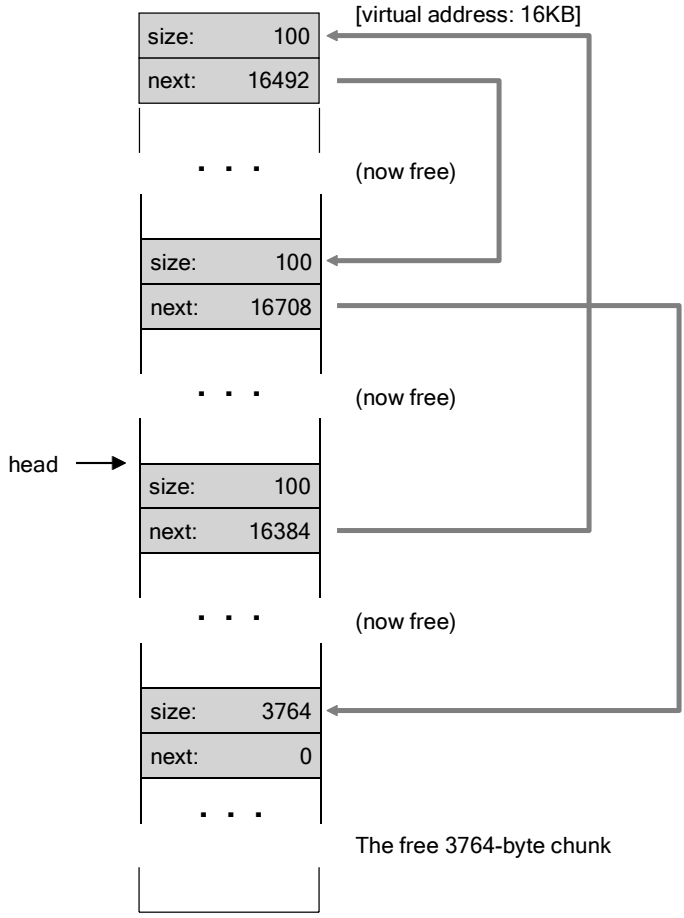


Figure 17.7: **Birleştirilmemiş Boş Bir Liste**

Şimdi küçük bir boş yığın (listenin başına işaret edilen 100 bayt) ve büyük bir boş yığın (3764 bayt) ile başlayan bir listemiz var. Listemizde sonunda birden fazla öğe var! Ve evet, boş alan parçalanmış, talihsiz ama yaygın bir olay.

Son bir örnek: Şimdi son iki kullanımda olan parçanın serbest bırakıldığını varsayalım. Birleşmeden parçalanma ile sonuçlanırsınız (Şekil 17.7).

Şekilden de görebileceğiniz gibi, şimdi büyük bir karmaşamız var! Niçin? Basit, listeyi **birleştirmeyi(coalesce)** unuttuk. Hafızanın tamamı boş olmasına rağmen parçalara ayrılır, böylece bir olmamasına rağmen parçalanmış bir hafıza olarak görünür. Çözüm basit: listeden geçin ve komşu parçaları **birleştirin(merge);** bittiğinde yığın tekrar bütün olacaktır.

Yığını Büyütmek

Birçok tahsis kütüphanesinde bulunan son bir mekanizmayı tartışmalıyız. Özellikle, yığın alanı biterse ne yapmalısınız? En basit yaklaşım sadece başarısız olmaktır. Bazı durumlarda bu tek seçenektir ve bu nedenle NULL döndürmek onurlu bir yaklaşımdır. Kendini kötü hissetme! Denedin ve başarısız olmana rağmen iyi dövüştün.

Geleneksel ayırıcıların çoğu küçük boyutlu bir yığınla başlar ve daha sonra işletim sistemi bittiğinde daha fazla bellek ister. Tipik olarak, bu, yığını büyütmek için bir tür sistem çağrısı yaptıkları (örneğin, çoğu UNİX sisteminde sbrk) ve ardından yeni parçaları oradan ayırdıkları anlamına gelir. Sbrk isteğine hizmet vermek için işletim sistemi boş fiziksel sayfalar bulur, bunları istekte bulunan işlemin adres alanına eşler ve ardından yeni yığının sonunun değerini döndürür; Bu noktada daha büyük bir yığın kullanılabilir ve istek başarıyla hizmet verilebilir

17.3 Temel Stratejiler

Artık kemerimizin altında bazı makinelerimiz olduğuna göre, boş alanı yönetmek için bazı temel stratejileri gözden geçirelim. Bu yaklaşımlar çoğunlukla kendiniz düşünebileceğiniz oldukça basit politikalara dayanmaktadır; Okumadan önce deneyin ve tüm alternatifleri (veya belki yenilerini) bulup bulmadığınızı görün!).

İdeal ayırıcı hem hızlıdır hem de parçalanmayı en aza indirir. Doğal olarak, tahsis akışı ve serbest istekler keyfi olabileceğinden (sonuçta bunlar programcı tarafından belirlenir), herhangi bir özel katman, yanlış girdi kümesi göz önüne alındığında oldukça kötü bir şekilde yapabilir. Bu nedenle, “en iyi" bir yaklaşımı tanımlamayacağız, bunun yerine bazı temel bilgiler hakkında konuşacağız ve artılarını ve eksilerini tartışacağız.

En uygun

En uygun(best fit) strateji oldukça basittir: ilk olarak, boş listede arama yapın ve istenen boyuttan büyük veya daha büyük boş bellek parçalarını bulun. Ardından, o aday grubundaki en küçük olanı iade edin; Bu, en uygun yığın olarak adlandırılır (buna en küçük uyum da denebilir). Geri dönecek doğru bloğu bulmak için ücretsiz listeden bir geçiş yeterlidir.

Best fit'in arkasındaki sezgi basittir: best fit, kullanıcının istediğine yakın bir blok döndürerek boşa harcanan alanı azaltmaya çalışır. Bununla birlikte, bir maliyet vardır; naif uygulamalar, doğru serbest blok için kapsamlı bir arama yaparken ağır bir performans cezası öder.

En kötü Uyum

En kötü uyum yaklaşımı, en iyi uyumun tam tersidir; en büyük parçayı bulun ve istenen miktarı iade edin; kalan (büyük) parçayı serbest listede tutun. En kötü uyum, böylece çok fazla yerine büyük parçaları serbest bırakmaya çalışır

en uygun yaklaşımdan kaynaklanabilecek küçük parçalar. Bununla birlikte, bir kez daha, tam bir boş alan araştırması gereklidir ve bu nedenle bu yaklaşım maliyetli olabilir. Daha da kötüsü, çoğu çalışma kötü performans gösterdiğini ve hala yüksek genel giderlere sahipken aşırı parçalanmaya yol açtığını gösteriyor.

İlk uyum

İlk sığdırma yöntemi(first fit), yeterince büyük olan ilk bloğu bulur ve istenen miktarı kullanıcıya döndürür. Daha önce olduğu gibi, kalan boş alan sonraki istekler için boş tutulur.

First fit'in hız avantajı vardır - tüm boş alanların kapsamlı bir şekilde aranmasına gerek yoktur - ancak bazen boş listenin başlangıcını küçük nesnelerle kirletir. Böylece, ayırıcının serbest listenin sırasını nasıl yönettiği bir sorun haline gelir. Bir yaklaşım, **adrese dayalı sıralamayı(address-based ordering)** kullanmaktır; Listeyi boş alanın adresine göre sıralı tutarak, birleştirme daha kolay hale gelir ve parçalanma azalma eğilimindedir.

Sonraki Uyum

Her zaman ilk sığdırma aramasını listenin başında başlatmak yerine, **sonraki sığdırma(first fit)** algoritması listedeki en son bakıldığı konuma fazladan bir işaretçi tutar. Buradaki fikir, boş alan aramalarını liste boyunca daha düzgün bir şekilde yaymak ve böylece listenin başlangıcının parçalanmasını önlemektir. Böyle bir yaklaşımın performansı, kapsamlı bir aramadan bir kez daha kaçınıldığı için first fit'e oldukça benzer.

Örnekler

İşte yukarıdaki stratejilerden birkaç örnek. Üzerinde 10, 30 ve 20 boyutlarında üç öğe bulunan ücretsiz bir liste düşünün (burada başlıkları ve diğer ayrıntıları görmezden geleceğiz, bunun yerine yalnızca stratejilerin nasıl işlediğine odaklanacağız):

15 Boyutunda bir ayırma isteği varsayalım. En uygun yaklaşım, tüm listeyi arayacak ve talebi karşılayabilecek en küçük boş alan olduğu için 20'nin en uygun olduğunu görecektir. Ortaya çıkan ücretsiz liste:

Bu örnekte olduğu gibi ve çoğu zaman en uygun yaklaşımla olduğu gibi, artık küçük bir serbest yığın kalmıştır. En uygun yaklaşım benzerdir, ancak bunun yerine en büyük parçayı bulur, bu örnekte 30. Ortaya çıkan liste

® 2008–19, ARPACI-DUSSEAU

Bu örnekte ilk uygun strateji, en uygun stratejiyle aynı şeyi yapar ve isteği karşılayabilecek ilk boş bloğu da bulur. Aradaki fark arama maliyetindedir; hem en uygun hem de en uygun olanı tüm listeye bakar; first-fit, yalnızca uygun olanı bulana kadar boş parçaları inceler, böylece arama maliyetini düşürür.

Bu örnekler yalnızca tahsis politikalarının yüzeyini çizer. Daha derin bir anlayış için gerçek iş yükleri ile daha ayrıntılı analizler ve daha karmaşık ayırıcı davranışlar (ör. Birleştirme) gereklidir. Belki ödev bölümü için bir şey, diyorsun?

17.4 Diğer Yaklaşımlar

Yukarıda açıklanan temel yaklaşımların ötesinde, bellek ayırmayı bir şekilde iyileştirmek için önerilen bir dizi teknik ve algoritma vardır. Bunlardan birkaçını burada değerlendirmeniz için listeliyoruz (yani, en uygun tahsisten biraz daha fazlasını düşünmenizi sağlamak için).

Ayrılmış Listeler

Bir süredir var olan ilginç bir yaklaşım, **ayrılmış listelerin(segregated lists)** kullanılmasıdır. Temel fikir basittir: Belirli bir uygulamanın yaptığı bir (veya birkaç) popüler boyutlu isteği varsa, yalnızca bu boyuttaki nesneleri yönetmek için ayrı bir liste tutun; Diğer tüm istekler daha genel bir bellek ayırıcıya iletilir.

Böyle bir yaklaşımın faydaları açıktır. Belirli bir istek boyutu için ayrılmış bir bellek parçasına sahip olmak, parçalanma çok daha az endişe vericidir; Ayrıca, bir listede karmaşık bir arama gerekmediğinden, ayırma ve ücretsiz istekler doğru boyutta olduklarında oldukça hızlı bir şekilde sunulabilir.

Her iyi fikir gibi, bu yaklaşım da bir sisteme yeni komplikasyonlar getirir. Örneğin, genel havuzun aksine, belirli bir boyuttaki özel isteklere hizmet eden bellek havuzuna ne kadar bellek ayrılmalıdır? Belirli bir ayırıcı, uber mühendisi Jeff Bonwick'in (Solaris çekirdeğinde kullanılmak üzere tasarlanmış) **levha ayırıcısı(slab allocator),** bu sorunu oldukça güzel bir şekilde ele alıyor [B94].

Özellikle, çekirdek önyüklendiğinde, sık sık istenmesi muhtemel çekirdek nesneleri için (kilitler, dosya sistemi inode'ları vb.) Bir dizi nesne önbelleği ayırır.); böylece **nesne önbelleklerinin(object caches)** her biri belirli bir boyutta ayrılmış boş listelerdir ve hızlı bir şekilde bellek ayırma ve boş istekler sunar. Belirli bir önbellekte boş alan azaldığında, daha genel bir bellek ayırıcısından bazı bellek plakaları ister (istenen toplam tutar, sayfa boyutunun ve söz konusu nesnenin katlarıdır). Tersine, belirli bir **levha(slabs**) içindeki nesnelerin referans sayılarının tümü sıfıra gittiğinde, genel ayırıcı bunları özel ayırıcıdan geri alabilir, bu genellikle VM sisteminin daha fazla belleğe ihtiyacı olduğunda yapılır.

BİR KENARA: BÜYÜK MÜHENDİSLER GERÇEKTEN HARİKA

Engineers like Jeff Bonwick (who not only wrote the slab allocator mentioned herein but also was the lead of an amazing file system, ZFS) are the heart of Silicon Valley. Behind almost any great product or technology is a human (or small group of humans) who are way above average in their talents, abilities, and dedication. As Mark Zuckerberg (of Facebook) says: "Someone who is exceptional in their role is not just a little better than someone who is pretty good. They are 100 times better." This is why, still today, one or two people can start a company that changes the face of the world forever (think Google, Apple, or Facebook). Work hard and you might become such a "100x" person as well. Failing that, work *with* such a person; you'll learn more in a day than most learn in a month. Failing that, feel sad.

Döşeme ayırıcısı, listelerdeki boş nesneleri önceden başlatılmış bir durumda tutarak çoğu ayrılmış liste yaklaşımının ötesine geçer. Bonwick, veri yapılarının başlatılmasının ve yok edilmesinin maliyetli olduğunu göstermektedir [B94]; Serbest bırakılan nesneleri belirli bir listede başlatılmış hallerinde tutarak, döşeme ayırıcısı böylece nesne başına sık başlatma ve imha döngülerinden kaçınır ve böylece genel giderleri belirgin şekilde düşürür.

Buddy Allocation

Birleştirme bir ayırıcı için kritik olduğundan, birleştirmeyi basitleştirmek için bazı yaklaşımlar tasarlanmıştır. İyi bir örnek, **ikili arkadaş ayırıcısında(binary buddy allocator)** [K65] bulunur.

Böyle bir sistemde, boş bellek ilk olarak kavramsal olarak 2 büyüklüğünde büyük bir alan olarak düşünülür. Bellek isteği yapıldığında, boş alan araması, isteği karşılayacak kadar büyük bir blok bulunana kadar boş alanı yinelemeli olarak ikiye böler (ve ikiye daha fazla bölünmesi çok küçük bir alana neden olur). Bu noktada istenen blok kullanıcıya iade edilir. İşte 7KB'LIK bir blok arayışında bölünen 64KB'LIK bir boş alan örneği (Şekil 17.8, sayfa 15).

Örnekte, en soldaki 8KB blok tahsis edilir (daha koyu gri tonuyla belirtildiği gibi) ve kullanıcıya iade edilir; Yalnızca iki boyutlu blokların gücünü vermenize izin verildiğinden, bu şemanın **dahili parçalanmadan(internal fragmentation)** muzdarip olabileceğini unutmayın.

Arkadaş tahsisinin güzelliği, o blok serbest bırakıldığında olanlarda bulunur. 8KB bloğunu boş listeye döndürürken, ayırıcı "arkadaş" 8kb'nin boş olup olmadığını kontrol eder; öyleyse, iki bloğu 16KB'LIK bir blokta birleştirir. Ayırıcı daha sonra 16KB bloğunun arkadaşının hala boş olup olmadığını kontrol eder; eğer öyleyse, bu iki bloğu birleştirir. Bu özyinelemeli birleştirme işlemi, tüm boş alanı geri yükleyerek veya bir arkadaşın kullanımda olduğu tespit edildiğinde durarak ağaçta devam eder.

® 2008–19, ARPACI-DUSSEAU

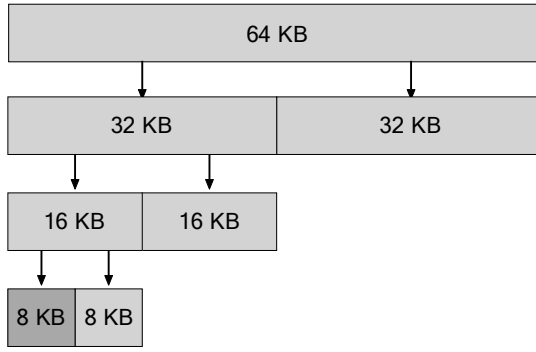


Figure 17.8: **Example Buddy-managed Heap**

Arkadaş tahsisinin bu kadar iyi çalışmasının nedeni, belirli bir bloğun arkadaşını belirlemenin basit olmasıdır. Nasıl, sordun mu? Yukarıdaki boş alandaki blokların adreslerini düşünün. Yeterince dikkatli düşünürseniz, her bir arkadaş çiftinin adresinin yalnızca tek bir bit ile farklılık gösterdiğini görürsünüz; hangi bit, arkadaş ağacındaki seviyeye göre belirlenir. Ve böylece ikili arkadaş ayırma şemalarının nasıl çalıştığına dair temel bir fikriniz var. Her zaman olduğu gibi daha fazla ayrıntı için Wilson anketine bakın [W + 95].

Other Ideas

Yukarıda açıklanan yaklaşımların çoğuyla ilgili önemli bir sorun, **ölçeklendirme (scaling)**eksikliğidir. Özellikle, arama listeleri oldukça yavaş olabilir. Bu nedenle, gelişmiş ayırıcılar bu maliyetleri karşılamak için daha karmaşık veri yapıları kullanır ve performans için basitlik ticareti yapar. Örnekler arasında dengeli ikili ağaçlar, yayvan ağaçlar veya kısmen sıralı ağaçlar bulunur [W + 95].

Modern sistemlerin genellikle birden fazla işlemciye sahip olduğu ve çok iş parçacıklı iş yükleri çalıştırdığı göz önüne alındığında (Eşzamanlılık kitabının bölümünde ayrıntılı olarak öğreneceğiniz bir şey), ayırıcıların çok işlemcili sistemlerde iyi çalışması için çok çaba harcanması şaşırtıcı değildir. tabanlı sistemler. Berger ve ark.'da iki harika örnek bulunur. [B + 00] ve Evans [E06]; ayrıntılar için onları kontrol edin.

Bunlar, insanların zaman içinde bellek ayırıcılar hakkında sahip oldukları binlerce fikirden sadece ikisi; Merak ediyorsanız kendi başınıza okuyun. Aksi takdirde, size gerçek dünyanın nasıl bir şey olduğu hakkında bir fikir vermek için glibc ayırıcısının [S15] nasıl çalıştığını okuyun.

17.5 Summary

Bu bölümde, mem- ory ayırıcılarının en temel biçimlerini tartıştık. Bu tür ayırıcılar, yazdığınız her C programına ve ayrıca kendi veri yapıları için mem- oriyi yöneten temel işletim sistemine bağlı olarak her yerde bulunur. Birçok sistemde olduğugibi,birçoksistemvardır

böyle bir sistemin oluşturulmasında yapılacak değiş tokuşlar ve bir tahsisatçıya sunulan tam iş yükü hakkında ne kadar çok şey bilirseniz, o iş yükü için daha iyi çalışacak şekilde ayarlamak için o kadar çok şey yapabilirsiniz. Çok çeşitli iş yükleri için iyi çalışan hızlı, alan açısından verimli, ölçeklenebilir bir ayırıcı yapmak, modern bilgisayar sistemlerinde süregelen bir zorluk olmaya devam etmektedir.

® 2008–19, ARPACI-DUSSEAU

References

[B+00] "Hoard: A Scalable Memory Allocator for Multithreaded Applications" by Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, Paul R. Wilson. ASPLOS-IX, November 2000. *Berger and company's excellent allocator for multiprocessor systems. Beyond just being a fun paper, also used in practice!*

[B94] "The Slab Allocator: An Object-Caching Kernel Memory Allocator" by Jeff Bonwick. USENIX '94. *A cool paper about how to build an allocator for an operating system kernel, and a greatexample of how to specialize for particular common object sizes.*

[E06] "A Scalable Concurrent malloc(3) Implementation for FreeBSD" by Jason Evans. April, 2006. http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf. *A detailed look at how to build a real modern allocator for use in multiprocessors. The "jemalloc" allocator is in widespread use today, within FreeBSD, NetBSD, Mozilla Firefox, and within Facebook.*

[K65] "A Fast Storage Allocator" by Kenneth C. Knowlton. Communications of the ACM, Volume 8:10, October 1965. *The common reference for buddy allocation. Random strange fact: Knuthgives credit for the idea not to Knowlton but to Harry Markowitz, a Nobel-prize winning economist. Another strange fact: Knuth communicates all of his emails via a secretary; he doesn't send email himself, rather he tells his secretary what email to send and then the secretary does the work of emailing.*

Last Knuth fact: he created TeX, the tool used to typeset this book. It is an amazing piece of software[4].

[S15] "Understanding glibc malloc" by Sploitfun. February, 2015. sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/. *A deep dive into how glibc malloc works. Amazingly detailed and a very cool read.*

[W+95] "Dynamic Storage Allocation: A Survey and Critical Review" by Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles. International Workshop on Memory Management, Scotland, UK, September 1995. *An excellent and far-reaching survey of many facets of memory allocation. Far too much detail to go into in this tiny chapter!*

[4]Actually we use LaTeX, which is based on Lamport's additions to TeX, but close enough.

Ödev (Simülasyon)

Program, malloc.py , bölümde açıklandığı gibi basit bir boş alan ayırıcısının davranışını keşfetmenizi sağlar. Temel işlemiyle ilgili ayrıntılar için BENİOKU bölümüne bakın.

Sorular

1. Birkaç rastgele ayırma ve serbest bırakma oluşturmak için önce -n 10 -H 0 -p BEST -s 0 bayraklarıyla çalıştırın. Alloc()/free() öğesinin ne döndüreceğini tahmin edebilir misiniz? Her istekten sonra ücretsiz listenin durumunu tahmin edebilir misiniz? Zaman içinde boş liste hakkında ne fark ediyorsunuz?

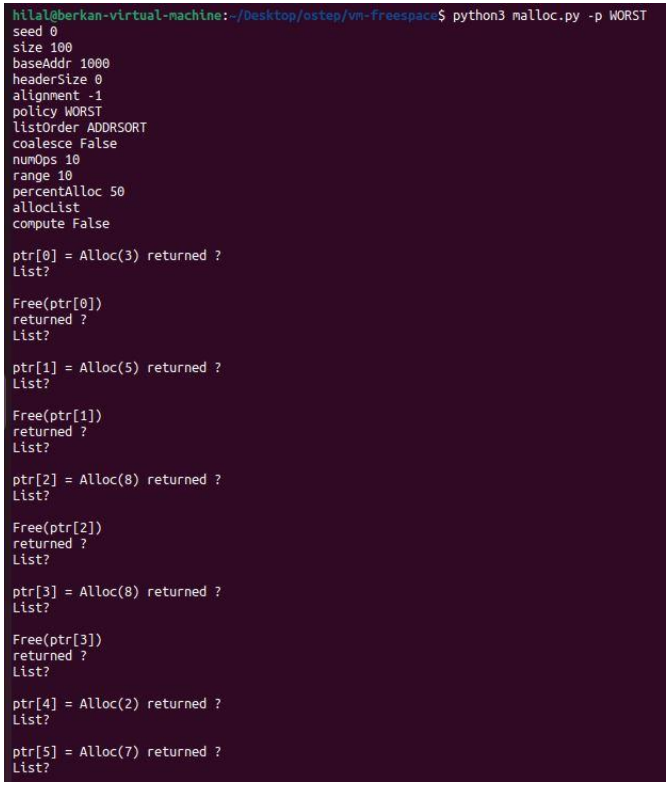
Hafıza parçalara ayrılır. Bu süreci çaşıştırdığımızda aşağıdaki gibi bir ekran görüntüsü oluşmaktadır

```
hilal@berkan-virtual-machine:~/Desktop/ostep/vm-freespace$ python3 malloc.py -n 10 -H 0 -p BEST -s 0
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute False

ptr[0] = Alloc(3) returned ?
List?

Free(ptr[0])
returned ?
List?

ptr[1] = Alloc(5) returned ?
List?

Free(ptr[1])
returned ?
List?

ptr[2] = Alloc(8) returned ?
List?

Free(ptr[2])
returned ?
List?

ptr[3] = Alloc(8) returned ?
List?

Free(ptr[3])
returned ?
List?

ptr[4] = Alloc(2) returned ?
List?

ptr[5] = Alloc(7) returned ?
List?
```

® 2008–19, ARPACI-DUSSEAU

2.Boş listeyi (-p EN KÖTÜSÜ) aramak için EN kötü uyum ilkesi kullanılırken sonuçlar nasıl farklıdır? Ne değişiyor?

Bellek daha fazla parçaya bölünür ve daha fazla öğe aranır.Bu süreci çalıştırdığımızda aşağıdaki gibi bir ekran görüntüsü oluşmaktadır.



3.FİRST fit (-p FİRST) kullanırken ne olur? First fit'i kullandığınızda ne hızlanır?

Daha az eleman arandı.Bu süreci çalıştırdığımızda aşağıdaki gibi bir ekran görüntüsü elde ederiz.

```
hilal@berkan-virtual-machine:~/Desktop/ostep/vm-freespace$ python3 malloc.py -p FIRST
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy FIRST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute False

ptr[0] = Alloc(3) returned ?
List?

Free(ptr[0])
returned ?
List?

ptr[1] = Alloc(5) returned ?
List?

Free(ptr[1])
returned ?
List?

ptr[2] = Alloc(8) returned ?
List?

Free(ptr[2])
returned ?
List?

ptr[3] = Alloc(8) returned ?
List?

Free(ptr[3])
returned ?
List?

ptr[4] = Alloc(2) returned ?
List?

ptr[5] = Alloc(7) returned ?
List?
```

® 2008–19, ARPACI-DUSSEAU

4.Yukarıdaki sorular için, listenin nasıl düzenli tutulduğu, bazı ilkeler için boş bir yer bulmak için gereken süreyi etkileyebilir. Farklı ücretsiz liste sıralamalarını kullanın (-merdiven sıralaması, -l BOYUT sıralaması +, --l BOYUTU SIRALAMA-) politikaların ve liste sıralamalarının nasıl etkileşime girdiğini görmek için.

$ ./malloc.py -p BEST -1 SIZESORT+ -c
$ ./malloc.py -p FIRST -1 SIZESORT+ -c
$ ./malloc.py -p WORST -1 SIZESORT- -c

```
hilal@berkan-virtual-machine:~/Desktop/ostep/vm-freespace$ python3 malloc.py -l ADDRSORT
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute False

ptr[0] = Alloc(3) returned ?
List?

Free(ptr[0])
returned ?
List?

ptr[1] = Alloc(5) returned ?
List?

Free(ptr[1])
returned ?
List?

ptr[2] = Alloc(8) returned ?
List?

Free(ptr[2])
returned ?
List?

ptr[3] = Alloc(8) returned ?
List?

Free(ptr[3])
returned ?
List?

ptr[4] = Alloc(2) returned ?
List?

ptr[5] = Alloc(7) returned ?
List?
```

```
hilal@berkan-virtual-machine:~/Desktop/ostep/vm-freespace$ python3 malloc.py -l SIZ
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder SIZESORT+
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute False

ptr[0] = Alloc(3) returned ?
List?

Free(ptr[0])
returned ?
List?

ptr[1] = Alloc(5) returned ?
List?

Free(ptr[1])
returned ?
List?

ptr[2] = Alloc(8) returned ?
List?

Free(ptr[2])
returned ?
List?

ptr[3] = Alloc(8) returned ?
List?

Free(ptr[3])
returned ?
List?

ptr[4] = Alloc(2) returned ?
List?

ptr[5] = Alloc(7) returned ?
List?
```

© 2008–19, ARPACI-DUSSEAU

```
hilal@berkan-virtual-machine:~/Desktop/ostep/vm-freespace$ python3 malloc.py -l SIZ
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder SIZESORT-
coalesce False
numOps 10
range 10
percentAlloc 50
allocList
compute False

ptr[0] = Alloc(3) returned ?
List?

Free(ptr[0])
returned ?
List?

ptr[1] = Alloc(5) returned ?
List?

Free(ptr[1])
returned ?
List?

ptr[2] = Alloc(8) returned ?
List?

Free(ptr[2])
returned ?
List?

ptr[3] = Alloc(8) returned ?
List?

Free(ptr[3])
returned ?
List?

ptr[4] = Alloc(2) returned ?
List?

ptr[5] = Alloc(7) returned ?
List?
```

5.Boş bir listenin birleştirilmesi oldukça önemli olabilir. Rastgele tahsislerin sayısını artırın (-n 1000'e söyleyin). Zaman içinde daha büyük tahsis taleplerine ne olur? Birleşerek ve birleşmeden çalıştırın (yani -C bayrağı olmadan ve -C bayrağıyla). Sonuç olarak ne gibi farklılıklar görüyorsunuz? Her durumda serbest liste zaman içinde ne kadar büyük? Bu durumda listenin sırası önemli mi?

$ ./malloc.py -n 1000 -r 30 -c
$ ./malloc.py -n 1000 -r 30 -c -C

Birleştirme olmadan, daha büyük ayırma istekleri NULL döndürür ve boş listenin boyutu daha büyüktür.

Adrese göre sıralamak daha iyidir.

```
hilal@berkan-virtual-machine:~/Desktop/ostep/vm-freespace$ python3 malloc.py -n 10 -C
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce True
numOps 10
range 10
percentAlloc 50
allocList
compute False

ptr[0] = Alloc(3) returned ?
List?

Free(ptr[0])
returned ?
List?

ptr[1] = Alloc(5) returned ?
List?

Free(ptr[1])
returned ?
List?

ptr[2] = Alloc(8) returned ?
List?

Free(ptr[2])
returned ?
List?

ptr[3] = Alloc(8) returned ?
List?

Free(ptr[3])
returned ?
List?

ptr[4] = Alloc(2) returned ?
List?

ptr[5] = Alloc(7) returned ?
List?
```

® 2008–19, ARPACI-DUSSEAU

6. Yüzde ayrılan kesir -P'yi 50'nin üzerine çıkardığınızda ne olur? 100'e yaklaştıkça tahsislere ne olur? Yüzde 0'a yaklaştığında ne olacak?

$ ./malloc.py -c -n 1000 -P 100
$ ./malloc.py -c -n 1000 -P 1

Ayrılacak yer kalmadı. Tüm işaretçiler serbest bırakılır.

```
hilal@berkan-virtual-machine:~/Desktop/ostep/vm-freespace$ python3 malloc.py -
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 0
allocList
compute False

Traceback (most recent call last):
  File "/home/hilal/Desktop/ostep/vm-freespace/malloc.py", line 186, in <modul
    assert(percent > 0)
AssertionError
```

```
hilal@berkan-virtual-machine:~/Desktop/ostep/vm-freespace$ python3 malloc.py -P 51
seed 0
size 100
baseAddr 1000
headerSize 0
alignment -1
policy BEST
listOrder ADDRSORT
coalesce False
numOps 10
range 10
percentAlloc 51
allocList
compute False

ptr[0] = Alloc(3) returned ?
List?

Free(ptr[0])
returned ?
List?

ptr[1] = Alloc(5) returned ?
List?

Free(ptr[1])
returned ?
List?

ptr[2] = Alloc(3) returned ?
List?

Free(ptr[2])
returned ?
List?

ptr[3] = Alloc(10) returned ?
List?

Free(ptr[3])
returned ?
List?

ptr[4] = Alloc(8) returned ?
List?

Free(ptr[4])
returned ?
List?
```

7. Oldukça parçalanmış bir boş alan oluşturmak için ne tür özel istekler yapabilirsiniz? Parçalanmış serbest listeler oluşturmak için -A bayrağını kullanın ve farklı ilke ve seçeneklerin serbest listenin organizasyonunu nasıl değiştirdiğini görün.

```
$ ./malloc.py -c -A +20,+20,+20,+20,+20,-0,-1,-2,-3,-4
$ ./malloc.py -c -A +20,+20,+20,+20,+20,-0,-1,-2,-3,-4  -C
$ ./malloc.py -c -A +10,-0,+20,-1,+30,-2,+40,-3  -1 SIZESORT-
$ ./malloc.py -c -A +10,-0,+20,-1,+30,-2,+40,-3  -1 SIZESORT-  -C
$ ./malloc.py -c -A +10,-0,+20,-1,+30,-2,+40,-3   -p FIRST  -1 SIZESORT+
$ ./malloc.py -c -A +10,-0,+20,-1,+30,-2,+40,-3   -P FIRST -1 SIZESORT+  -C
$ ./malloc.py -c -A +10,-0,+20,-1,+30,-2,+40,-3  -P FIRST -1 SIZESORT-
$ ./malloc.py -c -A +10,-0,+20,-1,+30,-2,+40,-3  -p FIRST -1 SIZESORT-  -C
$ ./malloc.py -c -A +10,-0,+20,-1,+30,-2,+40,-3  -p WORST -1 SIZESORT+
$ ./malloc.py -c -A +10,-0,+20,-1,+30,-2,+40,-3  -p WORST -1 SIZESORT+  -C
$ ./malloc.py -c -A +10,-0,+20,-1,+30,-2,+40,-3  -p WORST -1 SIZESORT-
$ ./malloc.py -c -A +10,-0,+20,-1,+30,-2,+40,-3  -p WORST -1 SIZESORT-  -C
```