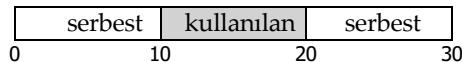


Serbest Alan Yönetimi

Bu bölümde, ister malloc kitaplığı (bir işlemin yiğinının sayfalarını yönetme) ister işletim sisteminin kendisi (adresin bölgümlerini yönetme) olsun, herhangi bir bellek yönetim sisteminin temel bir yönünü tartışmak için belleği sanallaştırma tartışmamızdan küçük bir sapma yapacağz, bir sürecin uzayı. Spesifik olarak, ***serbest Alan yönetimini(free-space management)*** çevreleyen konuları 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, sadece bu sabit boyutlu birimlerin bir listesini tutarsınız; bir istemci bunlardan birini talep ettiğinde, ilk giriş döndürürsünüz.

Bos alan yönetiminin daha zor (ve ilginc) hale geldiği yerler yönettiğiniz bos alan değişken boyutlu birimlerden oluşuyorsa; bu, kullanıcı düzeyinde bellek tahsis eden bir kütüphanede (malloc() ve free()'de olduğu gibi) ve sanal belleği uygulamak için ***segmentasyon(bölme)*** kullanırken fiziksel belleği yöneten bir işletim sisteminde ortaya çıkar. Her iki durumda da, var olan 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 olmadığı için sonraki istekler başarısız olabilir.



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

KONUNUN CAN ALICI NOKTASI: SERBEST ALAN NASIL YÖNETİLİR

Değişken boyutlu istekleri karşılarken FREE SPACE nasıl yönetilmelidir?
 Parçalanmayı en aza indirmek için hangi stratejiler kullanılabilir?
 Ne alternatif yaklaşımların zaman ve mekan ek yükleri nelerdir?

17.1 Varsayımlar

Bu tartışmanın çoğu, kullanıcı düzeyinde bellek ayırma kitaplıklarında bulunan ayırcıların harika geçmişine odaklanacaktır.

Wilson'in mükemmel anketinden [W+95] yararlanıyoruz, ancak ilgili okuyucuları daha fazla ayrıntı için kaynak belgeye gitmeye teşvik ediyoruz.

`malloc()` ve `free()` tarafından sağlanan gibi temel bir arabirim varsayıyoruz.

Özellikle, `void *malloc(size t size)` tek bir parametre alır, bu, uygulama tarafından istenen bayt sayısıdır; o boyuttaki (veya daha büyük) bir bölgeye bir işaretçi (belirli bir türde olmayan veya C dilinde bir *boş işaretçi(void pointer)*) geri verir. Tamamlayıcı rutin `void free(void *ptr)` bir işaretçi alır ve karşılık gelen öbeği serbest bırakır. Arayüzün ima ettiği şeye dikkat edin: kullanıcı, alanı boşaltırken kütüphane boyutu hakkında bilgilendirmez; bu nedenle, kütüphane, kendisine yalnızca bir işaretçi verildiğinde, bir bellek yiğininin ne kadar büyük olduğunu anlayabilmelidir. Bunu biraz sonra bölümde nasıl yapacağımızı tartışacağız.

Bu kütüphanenin yönettiği alan, tarihsel olarak *heap (yiğım)* olarak bilinir ve yiğindaki serbest alanı yönetmek için kullanılan genel veri yapısı, bir tür *serbest listedir(free list)*. Bu yapı, belleğin yönetilen bölgesindeki tüm boş alan yiğinlarına referanslar içerir. Tabii ki, bu veri yapısının kendi başına bir liste olması gerekmek, ancak boş alan izlemek için sadece bir tür veri yapısı olması gereklidir.

Ayrıca, yukarıda açıklandığı gibi, öncelikli olarak *dış parçalanma(external fragmentation)* ile ilgilendiğimizi varsayıyoruz. Ayırcılar, elbette *dahili parçalanma(internal fragmentation)* sorunu da yaşayabilirler; bir ayrıca, talep edilenden daha büyük bellek yiğinları dağıtırsa, böyle bir yiğindaki sorulmamış (ve dolayısıyla kullanılmayan) herhangi bir alan, dahili parçalanma olarak kabul edilir (çünkü atık, tahsis edilen birimin içinde gerçekleşir) ve alan israfına başka bir örnektr.

Bununla birlikte, basitlik adına ve iki tür parçalanmadan daha ilginç olduğu için, coğullukla dış parçalanmaya odaklanacağız.

Ayrıca, bellek istemciye dağıtıldıktan sonra, bellekte başka bir konuma taşınmayıcağıını da varsayıcağız. Örneğin, bir program `malloc()` ögesini çağırırsa ve yiğin içindeki bir alana işaretçi verilirse,

bu bellek bölgesi, program karşılık gelen bir `free()` çağrısı yoluyla döndürüne kadar esasen programa "aitir" (ve kitaplık tarafından taşınamaz).

¹Yaklaşık 80 sayfa uzunluğunda; bu nedenle gerçekten ilgilenmeniz gerekiyor!

Bu nedenle, parçalanmayla mücadelede faydalı olabilecek serbest alan **sıkıştırması**(compaction) mümkün değildir. Bununla birlikte sıkıştırma, işletim sisteminde **segmentasyonu(bölme)** uygularken parçalanma ile başa çıkmak için kullanabilir (segmentasyonla ilgili bölümde tartışıldığı gibi).

Son olarak, ayırıcıın bitişik bir bayt bölgesini yönettiğini varsayıcağız. Bazı durumlarda, bir ayırıcı o bölgenin büyümemesini isteyebilir; örneğin, kullanıcı düzeyinde bir bellek ayırma kütüphanesi, alanı dolduğunda yiğini büyütmek için çekirdeği arayabilir (sbrk gibi bir sistem çağrısı yoluyla). Ancak, basit olması için, bölgenin ömrü boyunca tek bir sabit boyutta olduğunu varsayıcağız.

17.2 Düşük Seviyeli Mekanizmalar

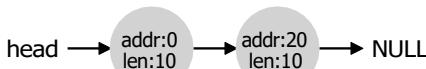
Bazı politika ayrıntılarına girmeden önce, çoğu ayırıcıda kullanılan bazı ortak mekanizmaları ele alacağız. İlk olarak, herhangi bir ayırıcıda yaygın olarak kullanılan teknikler olan bölme ve birleştirmenin temellerini tartışacağız. İkinci olarak, tahsis edilen bölgelerin boyutunun nasıl hızlı ve nispeten kolay bir şekilde takip edilebileceğini göstereceğiz. Son olarak, nelerin serbest olup olmadığını takip etmek için boş alan içinde nasıl basit bir liste oluşturacağımızı tartışacağız.

Bölme ve Birleştirme

Bos bir liste, yiğında kalan boş alanı tanımlayan bir dizi öğe içerir. Böylece, aşağıdaki 30 baytlık yiğini varsayıyalım:

serbest	kullanılan	serbest
0	10	20

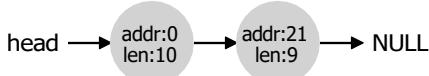
Bu yiğin için serbest listede iki öğe olacaktır. Bir giriş, ilk 10 baytlık boş bölümü (0-9 bayt) ve bir giriş diğer boş bölümü (20-29 bayt) açıklar:



Yukarıda açıklandığı gibi, 10 bayttan büyük herhangi bir istek, başarısız (NULL döndürür); bu boyutta kullanılır tek bir bitişik bellek parçası yok. Tam olarak bu boyuta (10 bayt) yönelik bir istek, serbest parçalardan herhangi biri tarafından kolayca karşılaşabilir. Ancak istek 10 bayttan küçük bir şey içinse ne olur?

²Bir C programına bir bellek parçasına bir işaretçi verdiğinizde, yürütmenin belirli bir noktasında diğer değişkenlerde ve hatta yazmaçlarda saklanabilecek olan o bölgeye yapılan tüm referansları (işaretçileri) belirlemek genellikle zordur. Bu durum, daha güçlü bir şekilde yazılan, çöp toplayan dillerde geçerli olmayıpabilir, bu nedenle parçalanma ile mücadele etmek için bir teknik olarak sıkıştırmayı mümkün kılacaktır.

Yalnızca tek bir bayt bellek talebimiz olduğunu varsayalım. Bunda durumda, ayırcı *bölme(splitting)* olarak bilinen bir eylemi gerçekleştirecektir: İsteği karşılayabilecek ve onu ikiye bölebilecek boş bir bellek parçası bulacaktır. İlk yoğun arayan kişiye geri dönecektir; ikinci parça listede kalacaktır. Bu nedenle, yukarıdaki örneğimizde, 1 bayt için bir talepte bulunulursa ve ayırcı, talebi karşılamak için listedeki iki öğeden ikincisini kullanmaya karar verirse, malloc() çağrıları 20 (1-byte'in adresi) döndürür. bayt ayrılmış bölge) ve liste şöyle görünür:



Resimde, listenin temelde bozulmadan kaldığını görebilirsiniz; tek değişiklik, serbest bölgenin artık 20 yerine 21'den başlaması ve bu serbest bölgenin uzunluğunun artık sadice 9³ olmasıdır. Bu nedenle, istekler belirli bir boş öbeğin boyutundan daha küçük olduğunda ayırma genellikle ayırcılarda kullanılır. Pek çok ayırcıda bulunan doğal bir mekanizma, serbest alanın *birleştirilmesi(coalescing)* olarak bilinir. Örneğimizi bir kez daha ele alalım (serbest 10 bayt, kullanılmış 10 bayt ve başka bir serbest 10 bayt).

Bu (küçük) yoğun göz önüne alındığında, bir uygulama free(10) öğesini çağrılarından ve böylece yoğunun ortasındaki boşluğu döndürdüğünde ne olur? Bu boş alanı çok fazla düşünmeden listemize geri eklesek, şuna benzeyen bir liste elde edebiliriz:



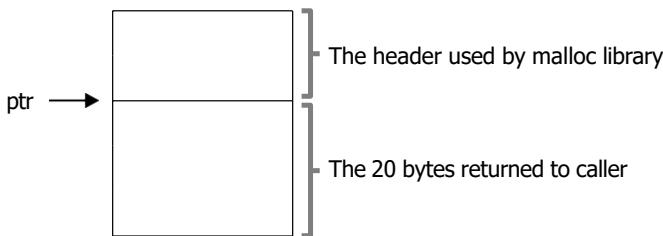
Soruna dikkat edin: yoğunun tamamı artık serbestken, görünüşe göre her biri 10 baylıklı üç parçaya bölünmüştür. Bu nedenle, bir kullanıcı 20 bayt isterse, basit bir liste geçisi bu kadar boş bir öbek bulamayacak ve başarısızlıkla sonuçlanacaktır.

Ayırcıların bu sorunu önlemek için yaptığı şey, bir yoğun bellek serbest bırakıldığında boş alanı birleştirmektir. Fikir basit: bellekte boş bir yoğun döndürürken, geri döndürdüğünüz yoğunun adreslerine ve yakındaki boş alan parçalarına dikkatlice bakın; yeni serbest bırakılan alan bir (veya bu örnekte olduğu gibi iki) mevcut serbest parçanın hemen yanında bulunuyorsa, bunları daha büyük tek bir boş yoğun halinde birleştirin. Böylece, birleştirme ile nihai listemiz şöyle görünmelidir:

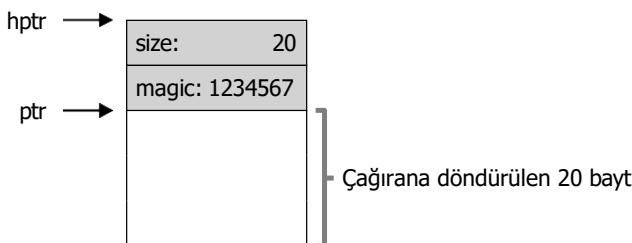


Gerçekten de, herhangi bir ayırma yapılmadan önce yoğun listesi ilk bakışta böyle görünüyordu. Birleştirme ile bir ayırcı, uygulama için geniş serbest uzantılarının kullanılabilir olmasını daha iyi sağlayabilir.

³Bu tartışma, gerçekçi olmayan ancak şimdilik basitleştirici bir varsayımdır. Başlıkların olmadığını varsayılmaktadır.



Şekil 17.1: Ayrılmış Bir Bölge ve Başlığı



Şekil 17.2: Başlığın Belirli İçerikleri

Tahsis Edilen Bölgelerin Büyüklüğünü İzleme

`free(void *ptr)` arabiriminin bir boyut parametresi almadığını fark etmiş olabilirsiniz; bu nedenle, bir işaretçi verildiğinde,

malloc kitaplığı, serbest bırakılan bellek bölgesinin boyutunu hızlı bir şekilde belirleyebilir ve böylece alanı tekrar boş listeye dahil edebilir.

Bu görevi gerçekleştirmek için, çoğu ayırcı, genellikle dağıtılan bellek öbeginden hemen önce, bellekte tutulan bir *ön etiket(header)* bloğunda 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 kullanıcının sonuçları `ptr`'de sakladığı düşünün, örn., `ptr = malloc(20);`

Ön etiket, ayrılan bölgenin boyutunu minimum düzeyde içerir (bunda

durum, 20); ayrıca serbest ayırmayı hızlandırmak için ek işaretçiler, ek bütünlük denetimi sağlamak için sıhırli bir sayı ve diğer bilgileri içerebilir. Bölgenin boyutunu ve bunun gibi bir sıhırli sayı içeren basit bir başlık varsayıyalım:

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

Yukarıdaki örnek, Şekil 17.2'de gördüğünüz gibi görünecektir. Kullanıcı `free(ptr)` öğesini çağrılığında, kütüphane başlığın nerede başladığını bulmak için basit işaretçi arıtmayı kullanır:

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

Başlığı yönelik böyle bir işaretçi elde ettikten sonra, kütüphane, sihirli sayının tutarlılık kontrolü (`assert(hptr->magic == 1234567)`) olarak beklenen değerle eşleşip eşleşmediğini kolayca belirleyebilir ve yeni serbest bırakılan bölgenin toplam boyutunu basit matematik ile hesaplayabilir. (yani, başlığın boyutunu bölgenin boyutuna eklemek). Son cümledeki küçük ama kritik ayrıntıya dikkat edin: serbest bölgenin boyutu, başlığın boyutu artı kullanıcının ayrılan alanın boyutudur. Bu nedenle, bir kullanıcı N bayt bellek istediginde, Kitaplık N boyutunda boş bir öbek aramaz; bunun yerine, N boyutunda ve başlığın boyutunda boş bir yiğin arar.

Serbest Liste Gömme

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

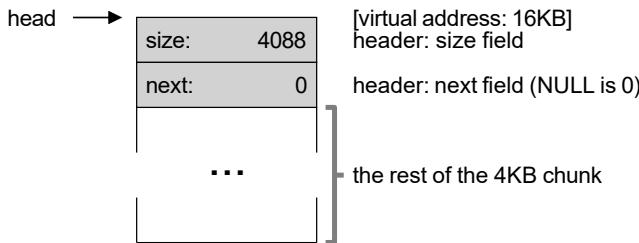
Daha tipik bir listede, yeni bir düğüm tahsis ederken, düğüm için alana ihtiyacınız olduğunda `malloc()` işlevini çağırırsınız. Ne yazık ki, bellek ayırma kitaplığında bunu yapamazsınız! Bunun yerine, listeyi boş alanın içinde oluşturmanız gereklidir. Bu biraz garip geliyorsa endişelenmeyin; öyle, ama yapamayacak kadar garip değil!

Yönetilecek 4096 baytlık bir bellek yiğinımız olduğunu varsayıyalım (yani, yiğin 4KB'dır). Bunu serbest bir liste olarak yönetmek için, önce söz konusu listeyi başlatmamız gereklidir; başlangıçta, listenin 4096 boyutundan (eksi başlık boyutu) bir girişi olmalıdır. İşte listedeki bir düğümün açıklaması:

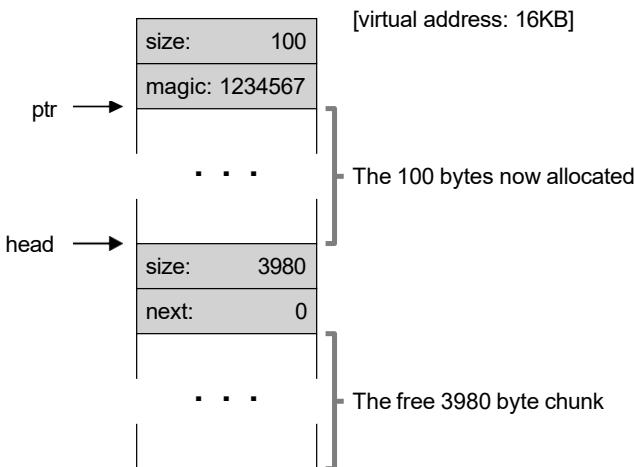
```
typedef struct __node_t {
    int           size;
    struct __node_t *next;
} node_t;
```

Şimdi yiğini başlatan ve boş listenin ilk öğesini bu boşluğun koyan bazı kodlara bakalım. Yiğinin `mmap()`; sistem çağrıları yoluyla elde edilen bir miktar boş alan içinde inşa edildiğini varsayıyoruz. Böyle bir yiğin oluşturulmanın tek yolu bu değil ama bu örnekte bize iyi hizmet ediyor. İşte kod:

```
// mmap() bir boş alan öbebine bir işaretçi döndürür
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;
```



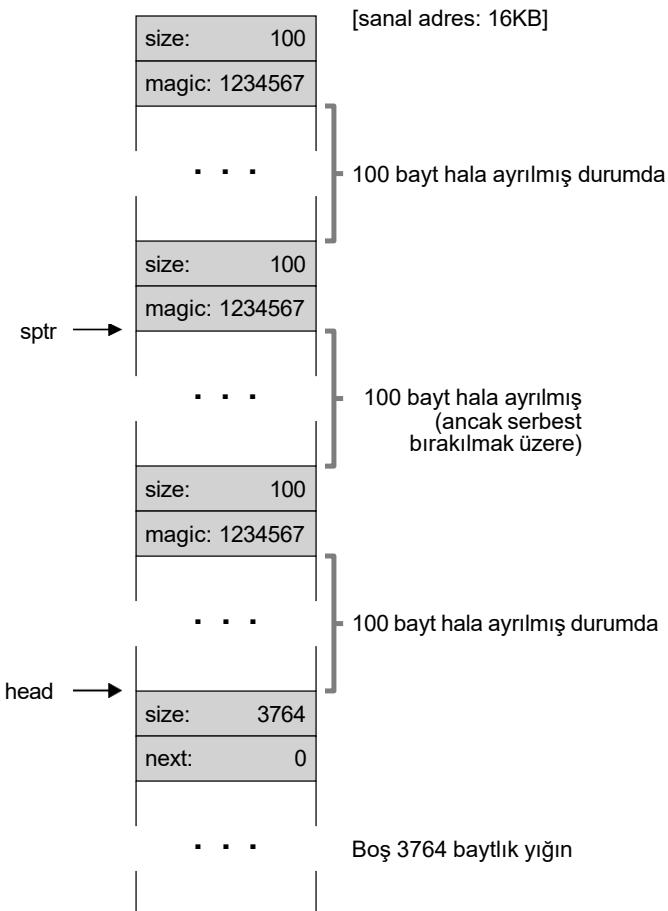
Şekil 17.3: Bir Boş Bölgesi Olan Yiğin



Şekil 17.4: Bir Yiğin: Bir Ayırımdan Sonra

Bu kodu çalıştırdıktan sonra listenin durumu, 4088 boyutunda tek bir girdiye sahip olmasıdır. Evet, bu küçük bir yiğin ama burada bizim için güzel bir örnek teşkil ediyor. Baş işaretçi, bu aralığın başlangıç adresini içerir; 16 KB olduğunu varsayıyalım (gerci herhangi bir sanal adres iyi olurdu). Böylece yiğin görsel olarak Şekil 17.3'te gördüğünüz gibi görünür.

Şimdi, diyelim ki 100 bayt boyutunda bir bellek yiğinının istendiğini düşünelim. Bu isteğe hizmet vermek için, kütüphane önce isteği karşılayacak kadar büyük bir parça bulacaktır; yalnızca bir boş parça olduğundan (boyut: 4088), bu yiğin seçilecektir. Daha sonra yiğin ikiye *bölünecektir(split)*: bir yiğin, isteğe hizmet edecek kadar büyük (ve yukarıda açıklandığı gibi başlık) ve kalan boş yiğin. 8 baytlık bir başlığı (bir tamsayı boyutu ve bir tamsayı sihirli sayı) varsayıarsak, yiğindaki boşluk artık Şekil 17.4'te gördüğünüz gibi görünür.

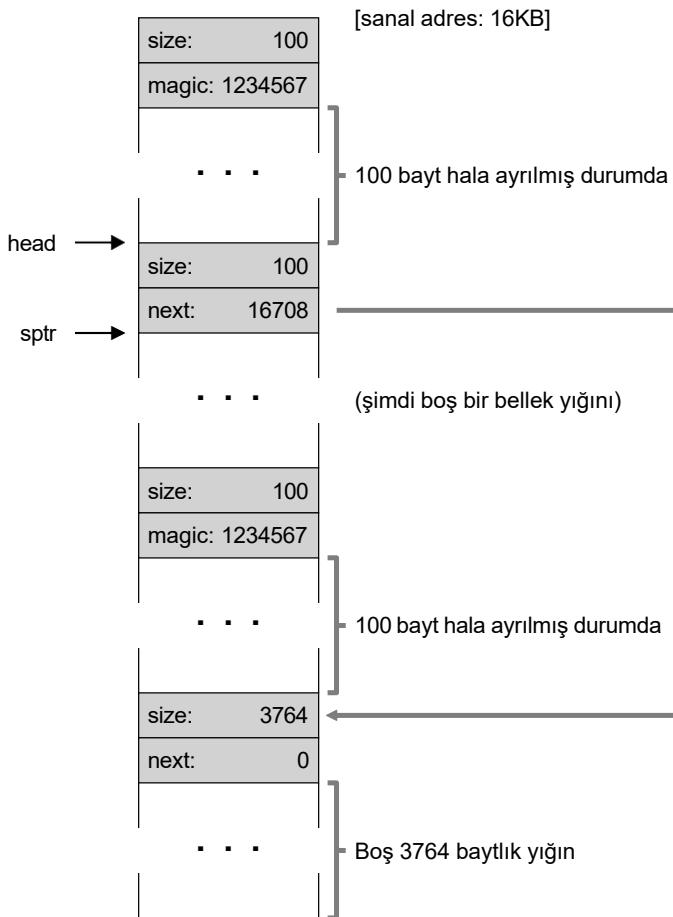


Şekil 17.5: Üç Yığın Ayrılmış Serbest Alan

Böylece, 100 baylıklı talep üzerine, kütüphane mevcut bir boş yığından 108 bayt ayırır, ona bir işaretçi döndürür (yukarıdaki şekilde ptr olarak işaretlenmiştir), başlık bilgisini, daha sonra kullanılmak üzere tahsis edilen alanın hemen önüne saklar. () ve listedeki bir boş düğümü 3980 bayta (4088 eksi 108) küçültür.

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

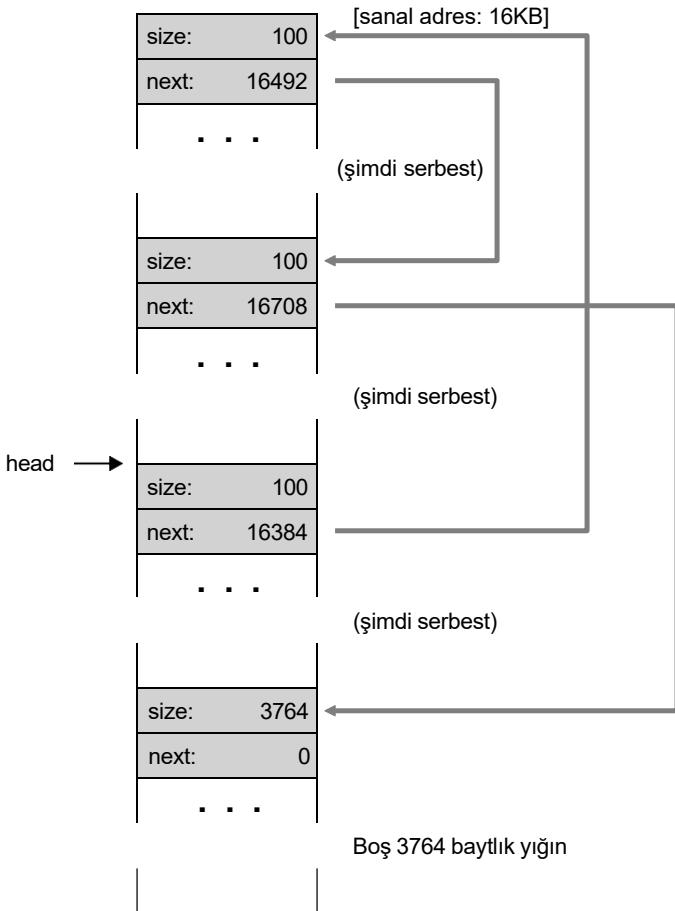
Orada görebileceğiniz gibi, yığının ilk 324 baytı artık tahsis edilmiştir ve bu nedenle, bu alanda üç başlık ve ayrıca çağrıran program tarafından kullanılan üç 100 baylıklı bölge görüyoruz. Serbest liste ilginç olmaya devam ediyor; yalnızca tek bir düğüm (baş tarafından işaret edildi), ancak şimdi üç bölmeden sonra yalnızca 3764 bayt boyutunda. Ancak, çağrıran program free() aracılığıyla bir miktar bellek döndürdüğünde ne olur?



Şekil 17.6: İki Yiğin Ayrılmış Boş Alan

Bu örnekte uygulama, `free(16500)` öğesini çağırarak ayrılan belleğin ortalığını döndürür (16500 değerine, bellek bölgesinin başlangıcı olan 16384, önceki yiğinin 108'ine ve 8 baytna eklenerek ulaşılır. bu parçanın başlığı). Bu değer önceki diyagramda `sprt` işaretçisi tarafından gösterilmiştir.

Kütüphane hemen serbest bölgenin boyutunu hesaplar ve ardından serbest öeği tekrar serbest listeye ekler. Boş listenin başına yerleştirdiğimizi varsayırsak, boşluk şuna benzer (Şekil 17.6).



Şekil 17.7: Birleştirilmemiş Serbest Liste

Şimdi küçük bir boş öbek (listenin başında gösterilen 100 bayt) ve büyük bir boş yiğin (3764 bayt) ile başlayan bir listemiz var. Sonunda listemizde birden fazla öğe var! Ve evet, boş alan parçalanmış, talihsiz ama yaygın bir olay.

Son bir örnek: Şimdi kullanımda olan son iki parçanın serbest kaldığını varsayıyalım. Birleşme olmazsa parçalanma olur (Şekil 17.7).

Şekilden de görebileceğiniz gibi, şimdi büyük bir karmaşa yaşıyoruz! Neden? Niye? Basit, listeyi *birleştirmemeyi*(*coalesce*) unuttuk. Hafızanın tamamı boş olmasına rağmen, parçalara ayrılmıştır, böylece bir olmamasına rağmen parçalanmış bir hafıza gibi görünür. Çözüm basit: listeyi gözden geçirin ve komşu parçaları birleştirin; bittiğinde, yiğin tekrar bütün olacaktır.

Yığını Büyütmek

Birçok tahsis kütüphanesinde bulunan son bir mekanizmayı tartışmamızıza. Özellikle, eğer yiğinda yer kalmazsa ne yapmalısınız? En basit yaklaşım sadece başarısız olmaktadır. Bazı durumlarda bu tek seçenekdir ve bu nedenle NULL döndürmek onurlu bir yaklaşımdır. Kendinizi kötü hissetmeyin! Nedeniniz ve başarısız olsanız da iyi bir mücadele verdiniz. Geleneğsel ayırcıların çoğu küçük boyutlu bir yiğinla başlar ve bittiğinde işletim sisteminden daha fazla bellek talep eder. Tipik olarak bu, yiğini büyütmek için bir tür sistem çağrıları (örneğin, çoğu UNIX sisteminde sbrk) yaptıkları ve ardından yeni parçaları oradan ayırdıkları anlamına gelir. Sbrk isteğine hizmet etmek için işletim sistemi boş fiziksel sayfalari bulur, bunları istekte bulunan sürecin adres alanına eşler ve ardından yeni yiğinin sonunun değerini döndürür; bu noktada daha büyük bir yiğin mevcuttur ve istek başarıyla karşılanabilir.

17.3 Basit Stratejiler

Artık elimizde biraz makine olduğuna göre, boş alanı yönetmek için bazı temel stratejilerin üzerinden geçelim. Bu yaklaşımlar çoğunlukla kendi başınıza düşünebileceğinizin oldukça basit politikalara dayanmaktadır; okumadan önce deneyin ve tüm alternatifleri (ya da belki bazı yenilerini!) bulup bulamayacağınızı görün.

İdeal ayırcı hem hızlidır hem de parçalanmayı en aza indirir. Ne yazık ki, tahsis ve serbest bırakma isteklerinin aksı keyfi olabileceğiinden (sonuçta programcı tarafından belirlenir), herhangi bir strateji yanlış girdiler kümesi verildiğinde oldukça kötü performans gösterebilir. Bu nedenle, "en iyi" yaklaşımı tanımlamak yerine bazı temel yaklaşımardan bahsedeecek ve bunların artı ve eksilerini tartışacağız.

BEST Fit(Uyum)

BEST fit(best fit) stratejisi oldukça basittir: ilk olarak, boş listede arama yapın ve istenen boyut kadar veya daha büyük olan boş bellek parçalarını bulun. Daha sonra, bu adaylar grubu içinde en küçük olanı döndürün; bu en uygun yiğin olarak adlandırılabilir (en küçük fit olarak da adlandırılabilir). Döndürülecek doğru bloğu bulmak için serbest listeden bir geçiş yeterlidir.

En iyi fit'in arkasındaki sezgi basittir: kullanıcının istediği yakını bir blok döndürerek, en iyi fit boş harcanan alanı azaltmaya çalışır. Ancak bunun bir maliyeti vardır; naif uygulamalar, doğru boş blok için kapsamlı bir arama gerçekleştirirken ağır bir performans cezası öder.

WORST Fit

WORST fit(worst fit) yaklaşımı, BEST fit'in tersidir; en büyük parçayı bulur ve istenen miktarı döndürür; kalan (büyük) parçayı serbest listede tutar. WORST fit böylece BEST fit yaklaşımından kaynaklanabilecek çok sayıda küçük parça yerine büyük parçaları serbest bırakmaya ve Ancak bir kez daha, boş alanın tamamının aranması gereklidir ve bu nedenle bu yaklaşım maliyetli olabilir.

Daha da kötüsü, çoğu çalışma bu yaklaşımın kötü performans gösterdiğini, aşırı parçalanmaya yol açarken yüksek ek yüklerle neden olduğunu göstermektedir.

FİRST Fit

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

FİRST fit hız avantajına sahiptir - tüm boş alanların kapsamlı bir şekilde aranması gerekmek - ancak bazen serbest listenin başlangıcını küçük nesnelerle kirletir. Bu nedenle, ayıricının serbest listenin sırasını nasıl yönettiği bir sorun haline gelir. Bir yaklaşım, *adres tabanlı sıralama(adress-based ordering)* kullanmaktadır; listeyi serbest alanın adresine göre sıralı tutarak, birleştirme daha kolay hale gelir ve parçalanma azalma eğilimindedir.

NEXT Fit

İlk arama işlemine her zaman listenin başından başlamak yerine, bir sonraki arama algoritması liste içinde en son aranan konuma fazladan bir işaretçi tutar. Buradaki fikir, boş alan aramalarını liste boyunca daha düzgün bir şekilde yapmak ve böylece listenin başlangıcının parçalanmasını önlemektir. Böyle bir yaklaşımın performansı, bir kez daha kapsamlı bir aramadan kaçınıldığı için ilk fit'e oldukça benzerdir.

Örnekler

İşte yukarıdaki stratejilere birkaç örnek. Üzerinde 10, 30 ve 20 boyutlarında üç öğe bulunan serbest bir liste hayal edin (burada başlıklarları ve diğer ayrıntıları görmezden geleceğiz, bunun yerine sadece stratejilerin nasıl işlediğine odaklanacağız):



Büyüklüğü 15 olan bir ayırmaya talebi olduğunu varsayıyalım. En iyi sonuç yaklaşımı tüm listeyi tarar ve talebi karşılayabilecek en küçük boş alan olduğunu için 20'nin en iyi sonuç olduğunu bulur. Ortaya çıkan serbest liste:



Bu örnekte olduğu gibi ve genellikle en iyi sonuç yaklaşımında olduğu gibi, artık küçük bir boş yığın kalmıştır. En kötü-fit yaklaşımı da benzerdir ancak bunun yerine en büyük yığını bulur, bu örnekte 30. Ortaya çıkan liste:



Bu örnekte first-fit stratejisi, en worst-fit ile aynı şeyi yapar, ayrıca isteği karşılayabilecek ilk serbest bloğu bulur. Aradaki fark arama maliyetindedir; hem en iyi-fit hem de en kötü-fit tüm listeye bakar; first-fit sadece uygun olanı bulana kadar serbest parçaları inceler, böylece arama maliyetini azaltır.

Bu örnekler ayıma politikalarının sadece yüzeyini çizmektedir. Daha derin bir anlayış için gerçek iş yükleri ve daha karmaşık ayırcı davranışları (örneğin, birleştirme) ile daha ayrıntılı analizler gereklidir. Belki de bir ev ödevi bölümü için bir şey dersiniz?

17.4 Diğer Yaklaşımlar

Yukarıda açıklanan temel yaklaşımların ötesinde, bellek ayırmayı bir şekilde iyileştirmek için bir dizi teknik ve algoritma önerilmiştir. Bunlardan birkaçını değerlendirmeniz için burada listeliyoruz (yani, sadece en iyi ayırmadan biraz daha fazlasını düşünmenizi sağlamak için).

Ayrıntılı Listeler

Bir süredir kullanılmakta olan ilginç bir yaklaşım da *ayrılmış listelerin* (*segregated list*) kullanılmasıdır. Temel fikir basittir: belirli bir uygulamanın yaptığı popüler boyutta bir (veya birkaç) isteği varsa, yalnızca bu boyuttaki nesneleri yönetmek için ayrı bir liste tutun; diğer tüm istekler daha genel bir bellek ayırcıya ilettilir. Böyle bir yaklaşımın faydalari açıkta. Belirli bir boyuttaki istekler için ayrılmış bir bellek yığınına sahip olarak, parçalanma çok daha az endişe vericidir; ayrıca, tahsis ve serbest bırakma istekleri doğru boyutta olduklarında oldukça hızlı bir şekilde sunulabilir, çünkü bir listenin karmaşık bir şekilde aranması gerekmektedir.

Her iyi fikir gibi bu yaklaşım da sisteme yeni komplikasyonlar getirmektedir. Örneğin, genel havuzun aksine, belirli bir boyuttaki özel isteklere hizmet eden bellek havuzuna ne kadar bellek ayrılmalıdır? Belirli bir ayırcı, über mühendisi Jeff Bonwick'in *tabaka ayırcısı* (*slab allocator*) (Solaris çekirdeğinde kullanılmak üzere tasarlanmıştır), bu sorunu oldukça güzel bir şekilde ele alır [B94].

Özellikle, çekirdek açıldığında, sıkça talep edilmesi muhtemel çekirdek nesneleri (kilitler, file sistemi inode'ları vb.) için bir dizi *nesne önbelleği* (*object caches*) tahsis eder; böylece nesne önbelleklerinin her biri belirli bir boyutta ayrılmış serbest listelerdir ve bellek ayırma ve serbest bırakma isteklerine hızlı bir şekilde hizmet eder. Belirli bir önbellekte boş alan azaldığında, daha genel bir bellek ayırcıdan bazı bellek dilimleri talep eder (talep edilen miktar sayfa boyutunu ve söz konusu nesnenin bir katıdır). Tersine, belirli bir tabaka içindeki nesnelerin referans sayıları sıfırındığında, genel ayırcı bunları özel ayırcıdan geri alabilir ve bu genellikle VM sistemi daha fazla belleğe ihtiyaç duyduğunda yapılır.

AYRICA: BÜYÜK MÜHENDISLER GERÇEKten BÜYÜKTÜR

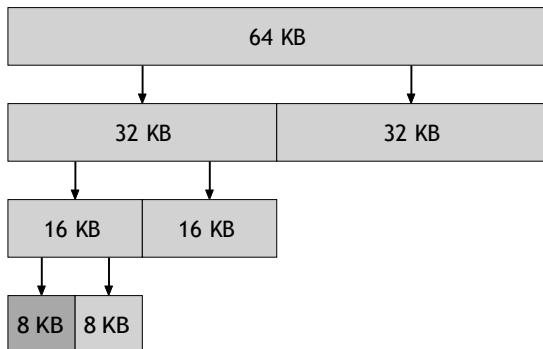
Jeff Bonwick gibi mühendisler (sadece burada bahsedilen tabaka ayırıcıyı yazmakla kalmayıp aynı zamanda muhteşem bir dosya sistemi olan ZFS'nin de liderliğini yapmıştır) Silikon Vadisi'nin kalbidir. Neredeyse her harika ürün ya da teknolojinin arkasında yetenekleri, becerileri ve adanmışlıklarıyla ortalamanın çok üzerinde olan bir insan (ya da küçük bir grup insan) vardır. Mark Zuckerberg'in (Facebook) dediği gibi: "Görevinde olağanüstü olan biri, oldukça iyi olan birinden sadece biraz daha iyi değildir. Onlar 100 kat daha iyidir." İşte bu nedenle bugün hala bir ya da iki kişi dünyanın cehresini sonsuza dek değiştiren bir şirket kurabiliyor (Google, Apple ya da Facebook'u düşünün). Çok çalışın ve siz de böyle bir "100x" kişi olabilirsiniz. Bunu başaramazsanız, böyle biriyle çalışın; çoğu kişinin bir ayda öğrendiğinden daha fazlasını bir günde öğrenirsiniz. Bunu başaramazsanız üzülün.

Tabaka ayırıcı ayrıca listelerdeki serbest nesneleri önceden başlatılmış bir durumda tutarak çoğu ayrılmış liste yaklaşımının ötesine geçer. Bonwick, veri yapılarının başlatılması ve yok edilmesinin maliyetli olduğunu göstermektedir [B94]; belirli bir listedeki serbest nesneleri başlatılmış durumda tutarak, tabaka ayırıcı böylece nesne başına sık başlatma ve yok etme döngülerinden kaçınır ve böylece ek yükleri belirgin şekilde azaltır.

Buddy(Arkadaş) Ayırıcı

Birleştirmenin bir ayırıcı için kritik olması nedeniyle, bazı yaklaşımlar birleştirimiği basitleştirmek üzere tasarlanmıştır. Bunun iyi bir örneği *ikili buddy ayırıcıda(binary buddy allocator)* bulunur [K65]. Böyle bir sistemde, boş bellek ilk olarak kavramsal olarak 2N boyutunda büyük bir alan olarak düşünülür. Bir bellek talebi yapıldığında, boş alan arayı, talebi karşılayacak kadar büyük bir blok bulunana kadar boş alanı tekrar ikiye böler (ve daha fazla ikiye bölmek çok küçük bir alanla sonuçlanır). Bu noktada, istenen blok kullanıcıya geri gönderilir. Aşağıda, 7KB'lık bir blok ararken 64KB'lık boş alanın bölünmesine bir örnek verilmiştir (Şekil 17.8, sayfa 15).

Örnekte, en soldaki 8KB'lık blok tahsis edilir (grinin daha koyu tonuyla gösterildiği gibi) ve kullanıcuya geri gönderilir; bu şemanın *dahili parçalanmadan(internal fragmentation)* muzdarip olabileceğini unutmayın, çünkü yalnızca ikiminin kuvveti büyülüğünde bloklar vermenize izin verilir. Buddy ayırmaların güzelliği, bu blok serbest bırakıldığında ne olacağından bulunur. Tahsis edici, 8KB'lık bloğu serbest listesine geri gönderirken, "arkadaş" 8KB'in serbest olup olmadığını kontrol eder; eğer öyleyse, iki bloğu 16KB'lık bir blocta birleştirir. Ayırıcı daha sonra 16KB'lık bloğun buddy'sinin hala boş olup olmadığını kontrol eder; boşsa bu iki bloğu birleştirir. Bu tekrarlayan birleştirme işlemi, ya tüm boş alanı geri yükleyerek ya da bir buddy'nin kullanımında olduğu tespit edildiğinde durarak ağaç boyunca devam eder.



Şekil 17.8: Buddy tarafından yönetilen Heap örneği

Buddy ayırmının bu kadar iyi çalışmasının nedeni, belirli bir bloğun buddy'sini tanımlamanın basit olmasıdır. Nasıl diye soruyorsunuz? Yukarıdaki boş alandaki blokların adreslerini düşünün. Yeterince dikkatli düşünürseniz, her bir buddy çiftinin adresinin yalnızca tek bir bit ile farklı olduğunu göreceksiniz; bu bit buddy ağacındaki seviye tarafından belirlenir. Ve böylece ikili buddy tahsis şemalarının nasıl çalıştığını dair temel bir fikriniz olur. Daha fazla ayrıntı için, her zaman olduğu gibi, Wilson araştırmasına [W+95] bakın.

Diger Fikirler

Yukarıda açıklanan birçok yaklaşımıyla ilgili önemli bir sorun *ölkelendirme(scaling)* eksikliğidir. Özellikle, listeleri aramak oldukça yavaş olabilir. Bu nedenle, gelişmiş ayırcılar bu maliyetleri karşılamak için daha karmaşık veri yapıları kullanır ve basitliği performansla değiştirir. Örnekler arasında dengeli ikili ağaçlar, yayılmış ağaçlar veya kısmen sıralı ağaçlar yer alır [W+95].

Modern sistemlerin genellikle birden fazla işlemciye sahip olduğu ve çok iş parçacıklı iş yükleri çalıştığı göz önüne alındığında (kitabın Eşzamanlılık bölümünde ayrıntılı olarak öğreneceksiniz), ayırcıların çok işlemci tabanlı sistemlerde iyi çalışması için çok çaba harcanması şaşırtıcı değildir. Berger ve diğerleri [B+00] ve Evans [E06]'da iki harika örnek bulunmaktadır; ayrıntılar için bunlara göz atın.

Burlar, insanların zaman içinde bellek ayırcılar hakkında ortaya attığı binlerce fikirden yalnızca ikisi; merak ediyorsanız kendiniz okuyun. Bunu yapamazsanız, gerçek dünyانın nasıl olduğunu anlamamanız için glibc ayırcının nasıl çalıştığını okuyun [S15].

17.5 Özet

Bu bölümde, bellek ayırıcıların en ilkel biçimlerini tartıştık. Bu tür ayırıcılar, yazdığınız her C programına bağlı olarak ve kendi veri yapıları için bellek yöneten temel işletim sisteminde her yerde mevcuttur. Pek çok sisteme olduğu gibi, böyle bir sistem oluştururken yapılması gereken pek çok fedakarlık vardır ve bir ayırıcıya 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 tasarruflu, ölçülebilir bir ayırıcı yapmak, modern bilgisayar sistemlerinde süregelen bir zorluk olmaya devam etmektedir.

Referanslar

- [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 great example 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: Knuth gives 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 software⁴.*
- [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!*

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

Ödev (Simülasyon)

malloc.py programı, bölümde açıklandığı gibi basit bir boş alan ayırıcının davranışını keşfetmenizi sağlar. Temel işleyişinin ayrıntıları için README'ye bakın.

Sorular

1. Önce birkaç rastgele tahsis ve serbest bırakma oluşturmak için -n 10 -H 0 -p BEST -s 0 bayraklarıyla(flagler ile) çalıştırın. alloc() / free() fonksiyonlarının ne döndürecekğini tahmin edebilir misiniz? Her istekten sonra serbest listenin durumunu tahmin edebilir misiniz? Zaman içinde serbest liste hakkında ne fark ettiniz?

./malloc.py -n 10 -H 0 -p BEST -s 0 -c komutunun çıktısı:

```

[n]
yekcan@yekcan: ~/Mesaiüstü/ostep/ostep-homework/vm-freespace
seed 0
size 100
baseaddr 1000
headaddr 0
alignment -1
maxsize 1000
listorder ABSORT
coulcece False
numops 10
range 10
percentAlloc 50
allocList
freeList
couplece True
ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1000 sz:97 ]

FreeList[0]
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ]

ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1008 sz:92 ]

FreeList[1]
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]

ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:84 ]

ptr[3] = Alloc(2) returned 1010 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1010 sz:84 ]

ptr[4] = Alloc(8) returned 1016 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[5] = Alloc(7) returned 1015 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1015 sz:1 ][ addr:1016 sz:84 ]

```

Zamanla, serbest liste büyür ve daha parçalı hale gelir. Özellikle de bir birleştirme olmadığı için bu durum böyle devam edecek.

Varsayılan parametrelerin çıktısı, istekle aynı boyutta veya daha büyük (en küçükü) boş bir blok döndüren en uygun strateji kullanılarak kullanılır ve alan birleştirilmeden geri kazanılır, böylece bölünmenin önündeki bloklar yeterli boyutta olmaz ve ancak daha sonra bulunabilir, bu da boş listenin önünde küçük parçalara neden olur.

2. Serbest listede arama yapmak için WORST fit ilkesi kullanıldığında sonuçlar nasıl değişir (-p WORST)? Ne gibi değişiklikler olur?

./malloc.py -n 10 -H 0 -p WORST -s 0 -c komutunun çıktısı:

```
ptr[0] = Alloc(0) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1000 sz:99 ]
ptr[1] = Alloc(0) returned 1000 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:99 ][ addr:1003 sz:97 ]
ptr[2] = Alloc(0) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:99 ][ addr:1000 sz:90 ]
ptr[3] = Alloc(0) returned 1000 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:99 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]
ptr[4] = Alloc(0) returned 1008 (searched 3 elements)
Free List [ Size 4 ]: [ addr:1000 sz:99 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]
ptr[5] = Alloc(0) returned 1016 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1000 sz:99 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1024 sz:76 ]
ptr[6] = Alloc(0) returned 1024 (searched 5 elements)
Free List [ Size 5 ]: [ addr:1000 sz:99 ][ addr:1003 sz:5 ][ addr:1016 sz:8 ][ addr:1024 sz:76 ][ addr:1020 sz:74 ]
ptr[7] = Alloc(0) returned 1020 (searched 5 elements)
Free List [ Size 5 ]: [ addr:1000 sz:99 ][ addr:1003 sz:5 ][ addr:1016 sz:8 ][ addr:1024 sz:76 ][ addr:1020 sz:74 ]
ptr[8] = Alloc(0) returned 1026 (searched 5 elements)
Free List [ Size 5 ]: [ addr:1000 sz:99 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1033 sz:67 ]
```

Serbest listeyi aramak için WORST Fit stratejisine geçildiğinde, WORST Fit en büyük boş parçayı bulmaya çalışır ve ardından yeterli alanı bölerek daha fazla küçük parçanın oluşturulmasına neden olur.

3.FIRST fit (-p FIRST) kullanıldığında ne olur? FIRST Fit'i kullandığınızda ne hızlanır?

/malloc.py -n 10 -H 0 -p FIRST -s 0 -c komutunun çıktısı:

```
root@yekcan:~/Masalıtı/ostep/ostep-homework/vm-freespace
yekcan@yekcan:~/Masalıtı/ostep/ostep-homework/vm-freespace
```

```
seed 0
size 100
baseAddr 1000
headerSize 4
alignment :1
policy FIRST
lcoation 10000000000000000000000000000000
coalesce False
numOps 10
range 0 10000000000000000000000000000000
percentageAlloc 50
allocList
```

```
ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1000 sz:9 ]
Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ]
ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1000 sz:92 ]
Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:9 ]
ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]
Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]
ptr[3] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]
Free(ptr[3])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]
ptr[4] = Alloc(2) returned 1000 (searched 1 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]
ptr[5] = Alloc(7) returned 1008 (searched 3 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1015 sz:1 ][ addr:1016 sz:84 ]
```

FIRST Fit stratejisi kullanıldığından, tahsis sonuçları BEST Fit ile aynıdır, ancak "BEST Fit" boyutlu bloğu döndürmeyi seçmek yerine eşleşir eşleşmez döndüğü için arama sayısı azalır. Ayrılan boyuta eşit veya daha büyük boyutta bir blok bulunarak döndürülür ve BEST Fit'e kıyasla arama sayısı önemli ölçüde azaltılır.

4. Yukarıdaki sorular için, listenin nasıl sıralandığı, bazı ilkeler için boş bir konum bulma süresini etkileyebilir. İlkelerin ve liste sıralamalarının nasıl etkileşime girdiğini görmek için farklı boş liste sıralamalarını (-1 ADDRSORT, -1 SIZESORT+, -1 SIZESORT-) kullanın.

Serbest listenin sıralanma şekli de aramanın verimliliğini etkileyebilir.

BEST Fit

BEST Fit, en iyi seçimi bulmak için tüm serbest listeyi dolaşmayı gerektirir, bu nedenle nasıl sıralandığına bakılmaksızın ayındır.

“Adrese göre sıralama”

./malloc.py -n 10 -H 0 -p BEST -l ADDRSORT -s 0 -c komutunun çıktısı:

```
root@yekcan:/home/yekcan/Downloads/osteap-homework/vm-freespace# ./malloc.py -n 10 -H 0 -p BEST -l ADDRSORT -s 0 -c
yekcan@yekcan:~/Masaüstü/osteap-homework/vm-freespace
```

```
seed 0
size 100
baseaddr 1000
headerSize 0
alignment :1
listorder ADDRSORT
coalesce False
maxPages 1
range 10
percentAlloc 50
blockSize 100
compute True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]; [ addr:1003 sz:97 ]

Free(ptr[0])
returned 0
Free List [ Size 2 ]; [ addr:1000 sz:3 ][ addr:1003 sz:97 ]

ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]; [ addr:1000 sz:3 ][ addr:1008 sz:92 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]; [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]

ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]; [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

ptr[3] = Alloc(6) returned 1008 (searched 4 elements)
Free List [ Size 3 ]; [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

Free(ptr[3])
returned 0
Free List [ Size 4 ]; [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[4] = Alloc(2) returned 1008 (searched 4 elements)
Free List [ Size 4 ]; [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[5] = Alloc(7) returned 1008 (searched 4 elements)
Free List [ Size 4 ]; [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1015 sz:1 ][ addr:1016 sz:84 ]
```

“Artan büyülük sırasına göre sıralama”

./malloc.py -n 10 -H 0 -p BEST -l SIZESORT+ -s 0 -c komutunun çıktısı:

```
[n]                                     yekcan@yekcan: ~/Masailistii/ostep/ostep-homework/vm-freespace
seed 0
size 100
baseAddr 1000
headerSize 0
alignment 1
policy BEST
listOrder SIZESORT+
coalesce False
numOps 0
range 10
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 [searched 1 elements]
Free List [ Size 1 ]: [ addr:1000 sz:97 ]

ptr[0] = Alloc(3) returned 1000 [searched 1 elements]
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1000 sz:97 ]

ptr[1] = Alloc(5) returned 1003 [searched 2 elements]
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1000 sz:92 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:5 ][ addr:1003 sz:92 ]

ptr[2] = Alloc(8) returned 1008 [searched 3 elements]
Free List [ Size 3 ]: [ addr:1000 sz:5 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

ptr[2] = Alloc(8) returned 1008 [searched 3 elements]
Free List [ Size 4 ]: [ addr:1000 sz:5 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

ptr[3] = Alloc(8) returned 1008 [searched 4 elements]
Free List [ Size 4 ]: [ addr:1000 sz:5 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

ptr[3] = Alloc(8) returned 1008 [searched 4 elements]
Free List [ Size 5 ]: [ addr:1000 sz:5 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

ptr[4] = Alloc(2) returned 1000 [searched 4 elements]
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1008 sz:5 ][ addr:1016 sz:84 ]

ptr[4] = Alloc(2) returned 1000 [searched 4 elements]
Free List [ Size 5 ]: [ addr:1002 sz:1 ][ addr:1008 sz:5 ][ addr:1016 sz:84 ]

ptr[5] = Alloc(7) returned 1008 [searched 4 elements]
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1015 sz:1 ][ addr:1016 sz:84 ]
```

“Azalan büyülük sırasına göre sıralama”

./malloc.py -n 10 -H 0 -p BEST -l SIZESORT- -s 0 -c komutunun çıktısı:

```
[n]                                     yekcan@yekcan: ~/Masailistii/ostep/ostep-homework/vm-freespace
seed 0
size 100
baseAddr 1000
headerSize 0
alignment 1
policy BEST
listOrder SIZESORT-
coalesce False
numOps 0
range 10
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 [searched 1 elements]
Free List [ Size 1 ]: [ addr:1000 sz:97 ]

ptr[0] = Alloc(3) returned 1000 [searched 1 elements]
Free List [ Size 2 ]: [ addr:1000 sz:97 ][ addr:1000 sz:3 ]

ptr[1] = Alloc(5) returned 1003 [searched 2 elements]
Free List [ Size 2 ]: [ addr:1000 sz:92 ][ addr:1003 sz:3 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:92 ][ addr:1003 sz:3 ][ addr:1000 sz:3 ]

ptr[2] = Alloc(8) returned 1008 [searched 3 elements]
Free List [ Size 3 ]: [ addr:1016 sz:84 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1016 sz:84 ][ addr:1008 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

ptr[3] = Alloc(8) returned 1008 [searched 4 elements]
Free List [ Size 4 ]: [ addr:1016 sz:84 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

Free(ptr[3])
returned 0
Free List [ Size 5 ]: [ addr:1016 sz:84 ][ addr:1008 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

ptr[4] = Alloc(2) returned 1000 [searched 4 elements]
Free List [ Size 4 ]: [ addr:1000 sz:84 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

ptr[4] = Alloc(2) returned 1000 [searched 4 elements]
Free List [ Size 5 ]: [ addr:1000 sz:84 ][ addr:1003 sz:5 ][ addr:1000 sz:1 ]
```

WORST Fit

WORST Fit’de tüm serbest listeyi dolaşır, bu nedenle nasıl sıralandığına bakılmaksızın aynıdır.

“Adrese göre sıralama”

./malloc.py -n 10 -H 0 -p WORST -l ADDRSORT -s 0 -c komutunun çıktısı:

```
seed 1
size 100
baseAddr 1000
headerSize 16
alignment 1
policy WORST
listPolicy ADDRSORT
coalesce False
numDps 10
pageSz 10
percentAlloc 50
allocList
compact True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:3 ][ addr:1003 sz:97 ]
Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ]

ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1008 sz:92 ]
Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]

ptr[2] = Alloc(6) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]
Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[3] = Alloc(8) returned 1016 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1024 sz:76 ]
Free(ptr[3])
returned 0
Free List [ Size 5 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1024 sz:76 ]

ptr[4] = Alloc(2) returned 1024 (searched 5 elements)
Free List [ Size 5 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1026 sz:74 ]
Free(ptr[4])
returned 0
Free List [ Size 6 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1026 sz:74 ]

ptr[5] = Alloc(7) returned 1026 (searched 5 elements)
Free List [ Size 6 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1033 sz:67 ]
```

“Artan büyülü sırasına göre sıralama”

/malloc.py -n 10 -H 0 -p WORST -l SIZESORT+ -s 0 -c komutunun çıktısı:

```
yekcan@yekcan1-MacBook-Pro:~/Desktop$ ./malloc.py -n 10 -H 0 -p WORST -l SIZESORT+ -s 0 -c
seed 0
size 10
baseaddr 1000
headerSize 0
alignment .1
policy WORST
listorder SIZESORT+
coalesce False
numDps 10
range 0
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:97 ]
Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ]
ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ][ addr:1008 sz:92 ]
Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:95 ][ addr:1008 sz:92 ]
ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:95 ][ addr:1016 sz:84 ]
Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:95 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]
ptr[3] = Alloc(4) returned 1016 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:95 ][ addr:1008 sz:8 ][ addr:1024 sz:76 ]
Free(ptr[3])
returned 0
Free List [ Size 5 ]: [ addr:1000 sz:3 ][ addr:1003 sz:95 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1024 sz:76 ]
ptr[4] = Alloc(2) returned 1024 (searched 5 elements)
Free List [ Size 5 ]: [ addr:1000 sz:3 ][ addr:1003 sz:95 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1024 sz:74 ]
```

“Azalan büyülü sırasına göre sıralama”

/malloc.py -n 10 -H 0 -p WORST -l SIZESORT- -s 0 -c komutunun çıktısı:

```
yekcan@yekcan1-MacBook-Pro:~/Desktop$ ./malloc.py -n 10 -H 0 -p WORST -l SIZESORT- -s 0 -c
seed 0
size 10
baseaddr 1000
headerSize 0
alignment .1
policy WORST
listorder SIZESORT-
coalesce False
numDps 10
range 0
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:97 ]
Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1003 sz:97 ][ addr:1000 sz:3 ]
ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:92 ][ addr:1000 sz:3 ]
Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:92 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]
ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1016 sz:84 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]
Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1016 sz:84 ][ addr:1008 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]
ptr[3] = Alloc(4) returned 1016 (searched 4 elements)
Free List [ Size 4 ]: [ addr:1024 sz:76 ][ addr:1008 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]
Free(ptr[3])
returned 0
Free List [ Size 5 ]: [ addr:1024 sz:76 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]
ptr[4] = Alloc(2) returned 1024 (searched 5 elements)
Free List [ Size 5 ]: [ addr:1024 sz:76 ][ addr:1016 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]
ptr[5] = Alloc(7) returned 1026 (searched 5 elements)
Free List [ Size 5 ]: [ addr:1033 sz:67 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]
```

FIRST Fit

FIRST Fit, istenen boyuta eşit veya daha büyük bir boş blok bulur ve onu döndürür, bu nedenle boş blokların sıralaması aramanın hızını etkiler.

Tahsis boyutunu görmek ve rastgele karşılaştırmak için adres sırasına göre sıralayın.

Büyüklük sırasına göre sıralandığından, büyük bloklara yönelik taleplerin bulunması daha uzun süreler.

Azalan boyut sırasına göre düzenlendiğinde, boyutu karşılayıp karşılamadığını görmek için yalnızca ilk bloğun kontrol edilmesi gereklidir, bu da daha fazla sayıda küçük blok üretme eğilimindedir.

“Adrese göre sıralama”

./malloc.py -n 10 -H 0 -p FIRST -l ADDRSORT -s 0 -c komutunun çıktısı:

```
yekcan@yekcan:~/Masası/stü/ostep/ostep-homework/vm-freespace$ ./malloc.py -n 10 -H 0 -p FIRST -l ADDRSORT -s 0 -c
seed
size 100
baseAddr 1000
headerSize 0
alignment -1
poolSize 1000
listOrder ADDRSORT
coalesce False
numOps 10
range 10
pageAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:97 ]

Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ]

ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1008 sz:92 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]

ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[3] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

Free(ptr[3])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[4] = Alloc(2) returned 1000 (searched 1 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[5] = Alloc(7) returned 1008 (searched 3 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1015 sz:1 ][ addr:1016 sz:84 ]
```

“Artan büyülük sırasına göre sıralama”

./malloc.py -n 10 -H 0 -p FIRST -l SIZESORT+ -s 0 -c komutunun çıktısı:

```
yelcan@yelcan:~/Masastu/ostep/ostep-homework/Vm-freespace$ ./malloc.py -n 10 -H 0 -p FIRST -l SIZESORT+ -s 0 -c
seed 0
size 100
base 1000
headerSize 0
alignment 1
policy FIRST
listOrder SIZESORT+
coalesce False
numOps 10
random 0
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:97 ]

Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1003 sz:97 ]

ptr[1] = Alloc(5) returned 1003 (searched 2 elements)
Free List [ Size 2 ]: [ addr:1000 sz:3 ][ addr:1008 sz:92 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:92 ]

ptr[2] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[3] = Alloc(8) returned 1008 (searched 3 elements)
Free List [ Size 3 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1016 sz:84 ]

Free(ptr[3])
returned 0
Free List [ Size 4 ]: [ addr:1000 sz:3 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[4] = Alloc(2) returned 1008 (searched 1 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1008 sz:8 ][ addr:1016 sz:84 ]

ptr[5] = Alloc(7) returned 1008 (searched 3 elements)
Free List [ Size 4 ]: [ addr:1002 sz:1 ][ addr:1003 sz:5 ][ addr:1015 sz:1 ][ addr:1016 sz:84 ]
```

“Azalan büyülük sırasına göre sıralama”

./malloc.py -n 10 -H 0 -p FIRST -l SIZESORT- -s 0 -c ko mutunun çıktısı:

```
yelcan@yelcan:~/Masastu/ostep/ostep-homework/Vm-freespace$ ./malloc.py -n 10 -H 0 -p FIRST -l SIZESORT- -s 0 -c
seed 0
size 100
base 1000
headerSize 0
alignment 1
policy FIRST
listOrder SIZESORT-
coalesce False
numOps 10
random 0
percentAlloc 50
allocList
compute True

ptr[0] = Alloc(3) returned 1000 (searched 1 elements)
Free List [ Size 1 ]: [ addr:1003 sz:97 ]

Free(ptr[0])
returned 0
Free List [ Size 2 ]: [ addr:1003 sz:97 ][ addr:1000 sz:3 ]

ptr[1] = Alloc(5) returned 1003 (searched 1 elements)
Free List [ Size 2 ]: [ addr:1008 sz:92 ][ addr:1000 sz:3 ]

Free(ptr[1])
returned 0
Free List [ Size 3 ]: [ addr:1008 sz:92 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

ptr[2] = Alloc(8) returned 1008 (searched 1 elements)
Free List [ Size 3 ]: [ addr:1016 sz:84 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

Free(ptr[2])
returned 0
Free List [ Size 4 ]: [ addr:1016 sz:84 ][ addr:1008 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

ptr[3] = Alloc(8) returned 1016 (searched 1 elements)
Free List [ Size 4 ]: [ addr:1024 sz:76 ][ addr:1008 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

Free(ptr[3])
returned 0
Free List [ Size 5 ]: [ addr:1024 sz:76 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

ptr[4] = Alloc(2) returned 1024 (searched 1 elements)
Free List [ Size 5 ]: [ addr:1026 sz:74 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]

ptr[5] = Alloc(7) returned 1026 (searched 1 elements)
Free List [ Size 5 ]: [ addr:1033 sz:67 ][ addr:1008 sz:8 ][ addr:1016 sz:8 ][ addr:1003 sz:5 ][ addr:1000 sz:3 ]
```

5. Serbest bir listenin birleştirilmesi oldukça önemli olabilir. Rastgele ayırma sayısını artırın (örneğin -n 1000'e). Zaman içinde daha büyük ayırma isteklerine ne olur? Birleştirme ile ve birleştirmeden (yani -C bayrağı olmadan ve -C bayrağı ile) çalıştırın. Sonuçlarda ne gibi farklılıklar görünsünüz? Her durumda boş liste zaman içinde ne kadar büyür? Bu durumda listenin sıralaması önemli mi?

CEVAP:

Rastgele işlem sayısını artırmak (-n 1000), birleştirme olmadan kullanılması zor olan çok küçük miktarda serbest alan yaratır ve ayrıca serbest listenin uzunluğunun artmasına neden olarak geçiş verimsiz hale getirir.

SONUÇLAR:

SAYFALARCA SONUÇ ÇIKTIĞI İÇİN HEPİNİ KOYAMIYORUM.

6. Ayrılan yüzde kesri -P'yi 50'den daha yüksek bir değere değiştirdiğinizde ne olur? 100'e yaklaşıkça ayrılan paylara ne olur?

Peki ya yüzde 0'a yaklaşıkça?

CEVAP:

Geri vermeden ayırdığınız yüzde ne kadar yüksek olursa, komşu bir adresin ayrılması için o kadar az şansınız olur. Bu nedenle, birleştirme şansı azalır ve bellek daha parçalı hale gelir.

7. Yüksek oranda parçalanmış bir serbest alan oluşturmak için ne tür özel taleplerde bulunabilirsiniz? Parçalanmış serbest listeler oluşturmak için -A bayrağını(flag) kullanın ve farklı politikaların ve seçeneklerin serbest listenin organizasyonunu nasıl değiştirdiğini görün.

CEVAP:

Çok sayıda ayırma işlemi gerçekleştirebiliriz.

`./malloc.py -n 100 -p WORST -P 60 -c: list size 40`

`./malloc.py -n 100 -p FIRST -l SIZESORT- -P 60 -c: list size 33`

Komutlarını kullanabiliriz.