17

Serbest Alan Yönetimi

Bu bölümde, ister malloc kitaplığı (bir işlemin yığınının sayfalarını yönetme) ister işletim sisteminin kendisi (adresin bölümlerini yönetme) olsun, herhangi bir bellek yönetim sisteminin temel bir yönünü tartışmak için belleği sanallaştırma tartışmamızdan küçük bir sapma yapacağız. bir sürecin uzayı). Spesifik olarak, **s*erbest 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 girişi döndürürsünüz.

Boş alan yönetiminin daha zor (ve ilginç) hale geldiği yerler yönettiğiniz boş 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.

|  |  |  |
| --- | --- | --- |
| serbest | kullanılan | serbest |

0 10 20 30

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

1

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?

* 1. Varsayımlar

Bu tartışmanın çoğu, kullanıcı düzeyinde bellek ayırma kitaplıklarında

bulunan ayırıcıların harika geçmişine odaklanacaktır.

Wilson'ın 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 işaretçiyi (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 yığınının ne kadar büyük olduğunu anlayabilmelidir. Bunu biraz sonra bölümde nasıl yapacağımızı tartışacağız.

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

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

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

Ayrıca, bellek istemciye dağıtıldıktan sonra, bellekte başka bir konuma taşınamayacağını da varsayacağız. Örneğin, bir program malloc() öğesini çağırırsa ve yığın içindeki bir alana işaretçi verilirse,

bu bellek bölgesi, program karşılık gelen bir free() çağrısı yoluyla döndürene

kadar esasen programa “aittir” (ve kitaplık tarafından taşınamaz).

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

OPERATİNG SYSTEMS [VERSİON 1.01]

[WWW.OSTEP.ORG](http://www.ostep.org/)

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 kullanılabilir (segmentasyonla ilgili bölümde tartışıldığı gibi).

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

* 1. 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

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

|  |  |  |
| --- | --- | --- |
| serbest | kullanılan | serbest |

0 10 20 30

Bu yığın 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:

head NULL

addr:0 len:10

addr:20 len:10

Yukarıda açıklandığı gibi, 10 bayttan büyük herhangi bir istek,

başarısız (NULL döndürür); bu boyutta kullanılabilir tek bir bitişik bellek parçası yok. Tam olarak bu boyuta (10 bayt) yönelik bir istek, serbest parçalardan herhangi biri tarafından kolayca karşılanabilir. 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 olmayabilir, bu nedenle parçalanma ile mücadele etmek için bir teknik olarak sıkıştırmayı mümkün kılacaktır.

Qc 2008–19, ARPACI-DUSSEAU THREE EASY PİECES

Yalnızca tek bir bayt bellek talebimiz olduğunu varsayalım. Bunda

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

addr:0 len:10

addr:21 len:9

head NULL

Resimde, listenin temelde bozulmadan kaldığını görebilirsiniz; tek değişiklik, serbest bölgenin artık 20 yerine 21'den başlaması ve bu serbest bölgenin uzunluğunun artık sadece 9³ olmasıdır. Bu nedenle, istekler belirli bir boş öbeğin boyutundan daha küçük olduğunda ayırma genellikle ayırıcılarda kullanılır. Pek çok ayırıcıda 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) yığın göz önüne alındığında, bir uygulama free(10) öğesini çağırdığında ve böylece yığının ortasındaki boşluğu döndürdüğünde ne olur? Bu boş alanı çok fazla düşünmeden listemize geri eklersek, şuna benzeyen bir liste elde edebiliriz:

addr:10 len:10

addr:0 len:10

addr:20 len:10

head NULL

Soruna dikkat edin: yığının tamamı artık serbestken, görünüşe göre

her biri 10 baytlık üç parçaya bölünmüştür. Bu nedenle, bir kullanıcı 20 bayt isterse, basit bir liste geçişi bu kadar boş bir öbek bulamayacak ve başarısızlıkla sonuçlanacaktır.

Ayırıcıların bu sorunu önlemek için yaptığı şey, bir yığın bellek serbest bırakıldığında boş alanı birleştirmektir. Fikir basit: bellekte boş bir yığın döndürürken, geri döndürdüğünüz yığının 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 yığın halinde birleştirin. Böylece, birleştirme ile nihai listemiz şöyle görünmelidir:

addr:0 len:30

head NULL

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

3Bu tartışma, gerçekçi olmayan ancak şimdilik basitleştirici bir varsayım olan başlıkların olmadığını varsaymaktadır.

OPERATİNG SYSTEMS [VERSİON 1.01]

[WWW.OSTEP.ORG](http://www.ostep.org/)

ptr

The header used by malloc library

The 20 bytes returned to caller

### Şekil 17.1: Ayrılmış Bir Bölge ve Başlığı

hptr

magic: 1234567

20

size:

ptr

Çağırana döndürülen 20 bayt

**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 ayırıcı, genellikle dağıtılan bellek öbeğinden 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 sakladığını 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 sihirli bir sayı ve diğer bilgileri içerebilir. Bölgenin boyutunu ve bunun gibi bir sihirli sayı içeren basit bir başlık varsayalım:

typedef struct { int size;

int magic;

} header\_t;

cQ2008–19, ARPACI-DUSSEAU

Three Easy Pieces

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

void free(void \*ptr) {

header\_t \*hptr = (header\_t \*) ptr - 1;

...

Başlığa 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 kullanıcıya ayrılan alanın boyutudur. Bu nedenle, bir kullanıcı N bayt bellek istediğinde, kitaplık N boyutunda boş bir öbek aramaz; bunun yerine, N boyutunda ve başlığın boyutunda boş bir yığın arar.

## Serbest Liste Gömme

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 malloc() işlevini çağırırsınız. Ne yazık ki, bellek ayırma kitaplığında bunu yapamazsınız! Bunun yerine, listeyi boş alanın içinde oluşturmanız gerekir. Bu biraz garip geliyorsa endişelenmeyin; öyle, ama yapamayacak kadar garip değil!

Yönetilecek 4096 baytlık bir bellek yığınımız olduğunu varsayalım (yani, yığın 4KB'dir). Bunu serbest bir liste olarak yönetmek için, önce söz konusu listeyi başlatmamız gerekir; başlangıçta, listenin 4096 boyutunda (eksi başlık boyutu) bir girişi olmalıdır. İşte listedeki bir düğümün açıklaması:

typedef struct node\_t { int size;

struct node\_t \*next;

} node\_t;

Şimdi yığını başlatan ve boş listenin ilk öğesini bu boşluğa koyan bazı kodlara bakalım. Yığının mmap(); sistem çağrısı yoluyla elde edilen bir miktar boş alan içinde inşa edildiğini varsayıyoruz. bBöyle bir yığın oluşturmanın tek yolu bu değil ama bu örnekte bize iyi hizmet ediyor. İşte kod:

// mmap() bir boş alan öbeğine 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;

OPERATİNG SYSTEMS [VERSİON 1.01]

[WWW.OSTEP.ORG](http://www.ostep.org/)

head

[virtual address: 16KB] header: size field

header: next field (NULL is 0)

...

0

next:

4088

size:

the rest of the 4KB chunk

### Şekil 17.3: Bir Boş Bölgesi Olan Yığın

[virtual address: 16KB]

100

size:

magic: 1234567

. . .

. . .

0

next:

3980

size:

ptr



The 100 bytes now allocated

head

The free 3980 byte chunk

### Şekil 17.4: Bir Yığın: Bir Ayırımdan Sonra

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

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

Qc 2008–19, ARPACI-DUSSEAU THREE PİECES

E ASY

100 bayt hala ayrılmış durumda

100

size:

magic: 1234567

. . .

100

size:

magic: 1234567

. . .

100

size:

magic: 1234567

. . .

. . .

0

next:

3764

size:

sptr



100 bayt hala ayrılmış

(ancak serbest bırakılmak üzere)

100 bayt hala ayrılmış durumda

head

Boş 3764 baytlık yığın

OPERATİNG SYSTEMS [VERSİON 1.01]

### Şekil 17.5: Üç Yığın Ayrılmış Serbest Alan

Böylece, 100 baytlık 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 baytlık (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 çağıran program tarafından kullanılan üç 100 baytlık 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, çağıran program free() aracılığıyla bir miktar bellek döndürdüğünde ne olur?

[WWW.OSTEP.ORG](http://www.ostep.org/)

[sanal adres: 16KB]

100

size:

magic: 1234567

. . .

100

size:

next: 16708

. . .

. . .

. . .

Boş 3764 baytlık yığın

0

next:

3764

size:

magic: 1234567

100

size:

(şimdi boş bir bellek yığını)

100 bayt hala ayrılmış durumda

100 bayt hala ayrılmış durumda

head

sptr

### Şekil 17.6: İki Yığın Ayrılmış Boş Alan

Bu örnekte uygulama, free(16500) öğesini çağırarak ayrılan belleğin orta yığınını döndürür (16500 değerine, bellek bölgesinin başlangıcı olan 16384, önceki yığının 108'ine ve 8 baytına eklenerek ulaşılır. bu parçanın başlığı). Bu değer önceki diyagramda sptr işaretçisi tarafından gösterilmiştir.

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

Qc 2008–19, ARPACI-DUSSEAU

Three Easy Pieces

|  |
| --- |
| size: 100 |
| next: 16492 |
| . . . |

|  |
| --- |
|  |
| size: 100 |
| next: 16708 |
| . . . |

head

(şimdi serbest)

(şimdi serbest)

(şimdi serbest)

|  |
| --- |
|  |
| size: 100 |
| next: 16384 |
| . . . |

|  |
| --- |
|  |
| size: 3764 |
| next: 0 |
| . . . |

Boş 3764 baytlık yığın

### Şekil 17.7: Birleştirilmemiş Serbest Liste

Şimdi küçük bir boş öbek (listenin başında gösterilen 100 bayt) ve büyük bir boş yığın (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 varsayalım. Birleşme olmazsa parçalanma olur (Şekil 17.7).

OPERATİNG SYSTEMS [VERSİON 1.01]

Şekilden de görebileceğiniz gibi, şimdi büyük bir karmaşa yaşıyoruz! Neden? Niye? Basit, listeyi ***birleştirmeyi(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, yığın tekrar bütün olacaktır.

[WWW.OSTEP.ORG](http://www.ostep.org/)

### Yığını Büyütmek

Birçok tahsis kütüphanesinde bulunan son bir mekanizmayı tartışmalıyız. Özellikle, eğer yığında yer kalmazsa ne yapmalısınız? En basit yaklaşım sadece başarısız olmaktır. Bazı durumlarda bu tek seçenektir ve bu nedenle NULL döndürmek onurlu bir yaklaşımdır. Kendinizi kötü hissetmeyin! Denediniz ve başarısız olsanız da iyi bir mücadele verdiniz. Geleneksel ayırıcıların çoğu küçük boyutlu bir yığınla başlar ve bittiğinde işletim sisteminden daha fazla bellek talep eder. Tipik olarak bu, yığını büyütmek için bir tür sistem çağrısı (örneğin, çoğu UNIX sisteminde sbrk) yaptıkları ve ardından yeni parçaları oradan ayırdıkları anlamına gelir. Sbrk isteğine hizmet etmek için işletim sistemi boş fiziksel sayfaları bulur, bunları istekte bulunan sürecin adres alanına eşler ve ardından yeni yığının sonunun değerini döndürür; bu noktada daha büyük bir yığın mevcuttur ve istek başarıyla karşılanabilir.

* 1. **Basit Stratejiler**

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

İdeal ayırıcı hem hızlıdır hem de parçalanmayı en aza indirir. Ne yazık ki, tahsis ve serbest bırakma isteklerinin akışı keyfi olabileceğinden (sonuçta programcı tarafından belirlenir), herhangi bir strateji yanlış girdiler kümesi verildiğinde oldukça kötü performans gösterebilir. Bu nedenle, "en iyi" yaklaşımı tanımlamak yerine bazı temel yaklaşımlardan bahsedecek 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 yığın olarak adlandırılır (en küçük ﬁt olarak da adlandırılabilir). Döndürülecek doğru bloğu bulmak için serbest listeden bir geçiş yeterlidir.

En iyi ﬁt'in arkasındaki sezgi basittir: kullanıcının istediğine yakın bir blok döndürerek, en iyi ﬁt boşa 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 gerekir ve bu nedenle bu yaklaşım maliyetli olabilir. Three

Easy Pieces

cQ2008–19, ARPACI-DUSSEAU

Daha da kötüsü, çoğu çalışma bu yaklaşımın kötü performans gösterdiğini, aşırı parçalanmaya yol açarken yüksek ek yüklere 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 gerekmez - ancak bazen serbest listenin başlangıcını küçük nesnelerle kirletir. Bu nedenle, ayırıcının serbest listenin sırasını nasıl yönettiği bir sorun haline gelir. Bir yaklaşım, ***adres tabanlı sıralama(adress-based ordering)*** kullanmaktır; 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 yaymak 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 başlıkları ve diğer ayrıntıları görmezden geleceğiz, bunun yerine sadece stratejilerin nasıl işlediğine

odaklanacağız):

10 30

20

head NULL

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

10

30

5

head NULL

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:

10

15

20

head NULL

OPERATİNG SYSTEMS [VERSİON 1.01]

[WWW.OSTEP.ORG](http://www.ostep.org/)

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

* 1. 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).

## Ayrıştırılmış 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 ayırıcıya iletilir. Böyle bir yaklaşımın faydaları açıktır. 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 gerekmez.

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 ayırıcı, uber mühendisi Jeff Bonwick'in ***tabaka ayırıcısı(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 ayırıcıdan bazı bellek dilimleri talep eder (talep edilen miktar sayfa boyutunun ve söz konusu nesnenin bir katıdır). Tersine, belirli bir tabaka içindeki nesnelerin referans sayıları sıfıra indiğinde, genel ayırıcı bunları özel ayırıcıdan geri alabilir ve bu genellikle VM sistemi daha fazla belleğe ihtiyaç duyduğunda yapılır.

Three Easy Pieces

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 çehresini 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 birleştirmeyi 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 arayışı, talebi karşılayacak kadar büyük bir blok bulunana kadar boş alanı tekrar 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 kullanıcıya geri gönderilir; bu şemanın ***dahili parçalanmadan(internal fragmentation)*** muzdarip olabileceğini unutmayın, çünkü yalnızca ikinin kuvveti büyüklüğünde bloklar vermenize izin verilir. Buddy ayırmanın güzelliği, bu blok serbest bırakıldığında ne olacağında bulunur. Tahsis edici, 8KB'lık bloğu serbest listesine geri gönderirken, "arkadaş" 8KB'ın serbest olup olmadığını kontrol eder; eğer öyleyse, iki bloğu 16KB'lık bir blokta 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 kullanımda olduğu tespit edildiğinde durarak ağaç boyunca devam eder.

Operating Systems

[VERSİON 1.01] [WWW.OSTEP.ORG](http://www.ostep.org/)

32 KB

64 KB

16 KB

16 KB

8 KB 8 KB

32 KB

### Şekil 17.8: Buddy tarafından yönetilen Heap örneği

Buddy ayırmanın 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 çalıştığına dair temel bir fikriniz olur. Daha fazla ayrıntı için, her zaman olduğu gibi, Wilson araştırmasına [W+95] bakın.

## Diğer Fikirler

Yukarıda açıklanan birçok yaklaşımla ilgili önemli bir sorun ***ölçkelendirme(scaling)*** eksikliğidir. Özellikle, listeleri aramak oldukça yavaş olabilir. Bu nedenle, gelişmiş ayırıcılar 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 çalıştırdığı göz önüne alındığında (kitabın Eşzamanlılık bölümünde ayrıntılı olarak öğreneceksiniz), ayırıcıların çok işlemcili 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.

Bunlar, insanların zaman içinde bellek ayırıcılar hakkında ortaya attığı binlerce fikirden yalnızca ikisi; merak ediyorsanız kendiniz okuyun. Bunu yapamazsanız, gerçek dünyanın nasıl olduğunu anlamanız için glibc ayırıcının nasıl çalıştığını okuyun [S15].

cQ2008–19, ARPACI-DUSSEAU

Three Easy Pieces

* 1. **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 sistemde 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, ölçeklenebilir bir ayırıcı yapmak, modern bilgisayar sistemlerinde süregelen bir zorluk olmaya devam etmektedir.

OPERATİNG SYSTEMS [VERSİON 1.01]

[WWW.OSTEP.ORG](http://www.ostep.org/)

**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 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!*

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

Qc 2008–19, ARPACI-DUSSEAU THREE EASY PİECES

Ödev (Simülasyon)

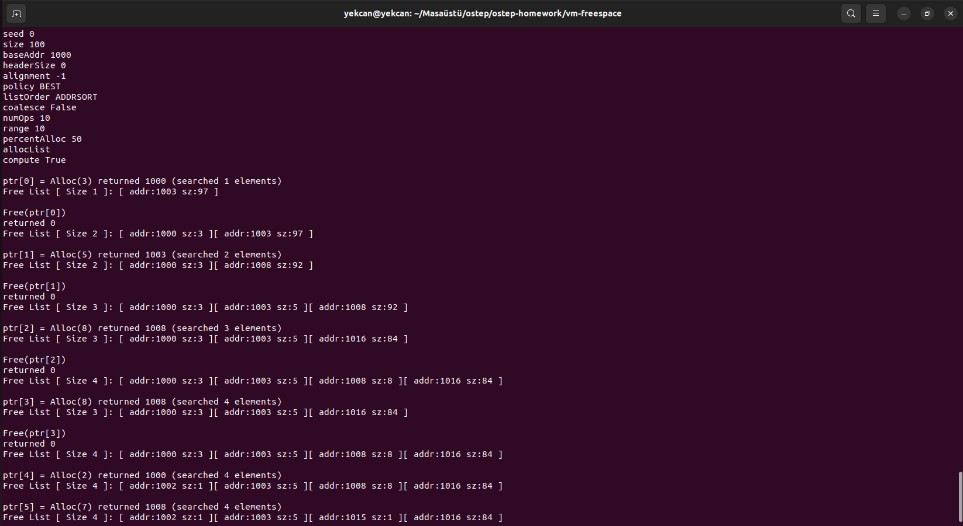
malloc.py programı, bölümde açıklandığı gibi basit bir boş alan ayırıcının davranışını keşfetmenizi sağlar. Temel işleyişinin ayrıntıları için README'ye bakın.

## Sorular

### 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 döndüreceğini 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:**



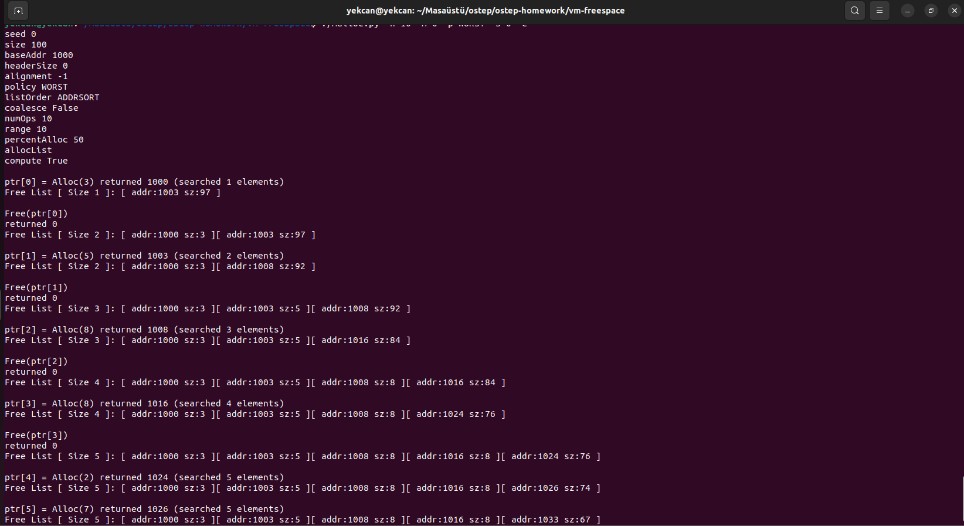
# Zamanla, serbest liste büyür ve daha parçalı hale gelir. Özellikle de bir birleştirme olmadığı için bu durum böyle devam edecek.

Varsayılan parametrelerin çıktısı, istekle aynı boyutta veya daha büyük (en küçüğü) 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.

## 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:**

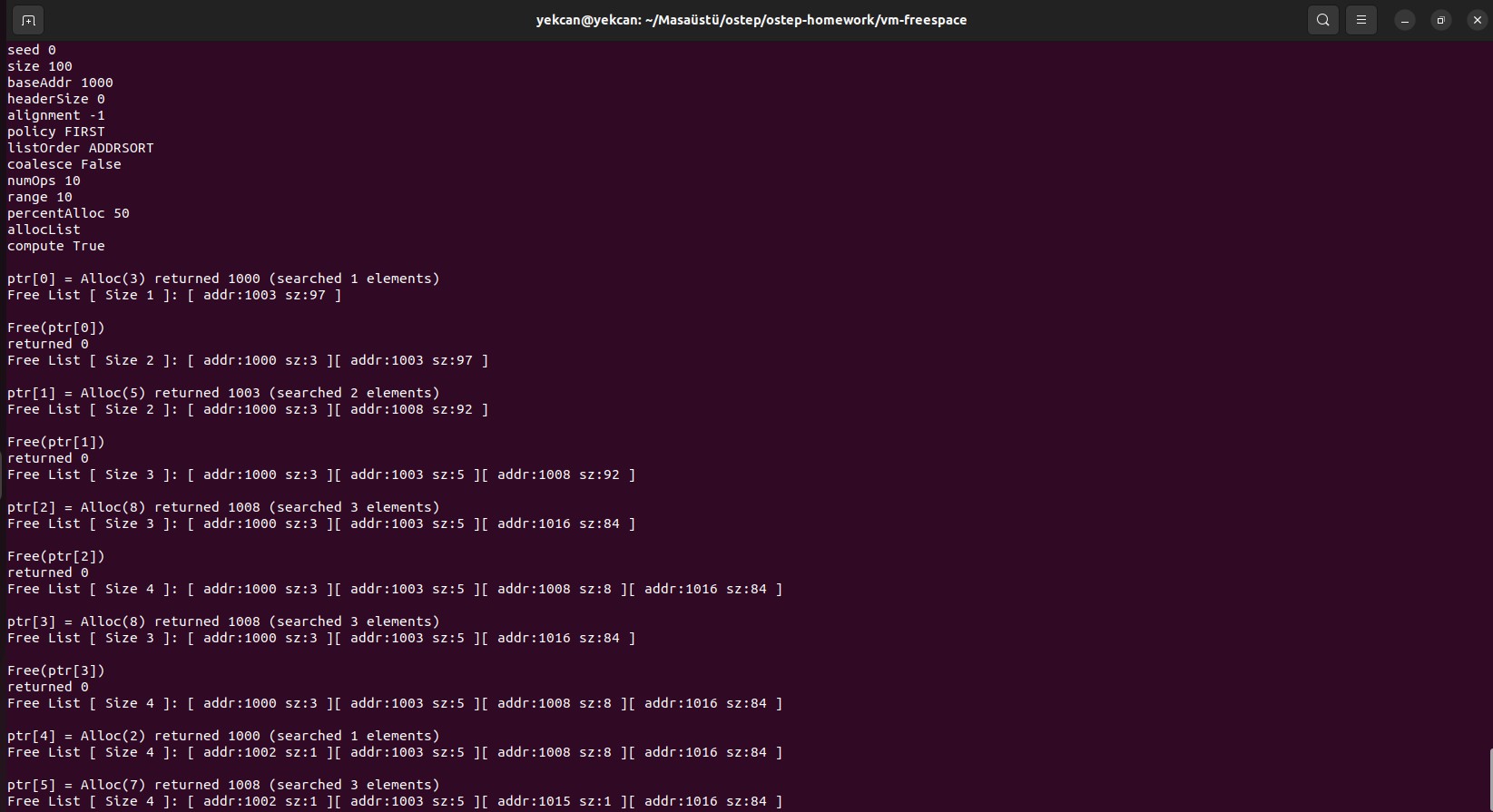


Serbest listeyi aramak için WORST Fit stratejisine geçildiğinde, WORST Fit en büyük boş parçayı bulmaya çalışır ve ardından yeterli alanı bölerek daha fazla küçük parçanın oluşturulmasına neden olur.

1. **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:**



FIRST Fit stratejisi kullanıldığında, 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.

1. **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 (-l ADDRSORT, -l SIZESORT+, -l SIZESORT-) kullanın.**

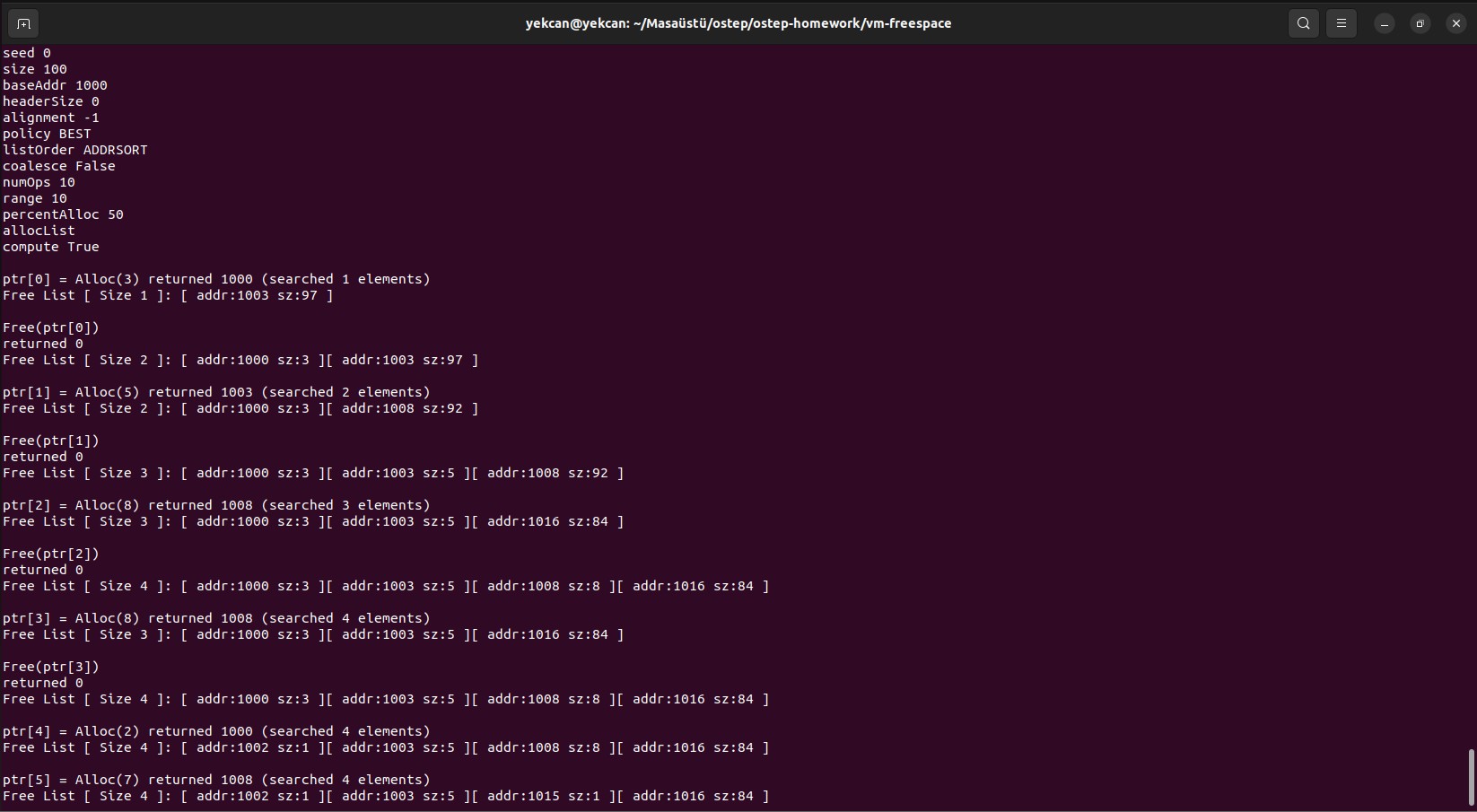
# Serbest listenin sıralanma şekli de aramanın verimliliğini etkileyebilir.

BEST Fit

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

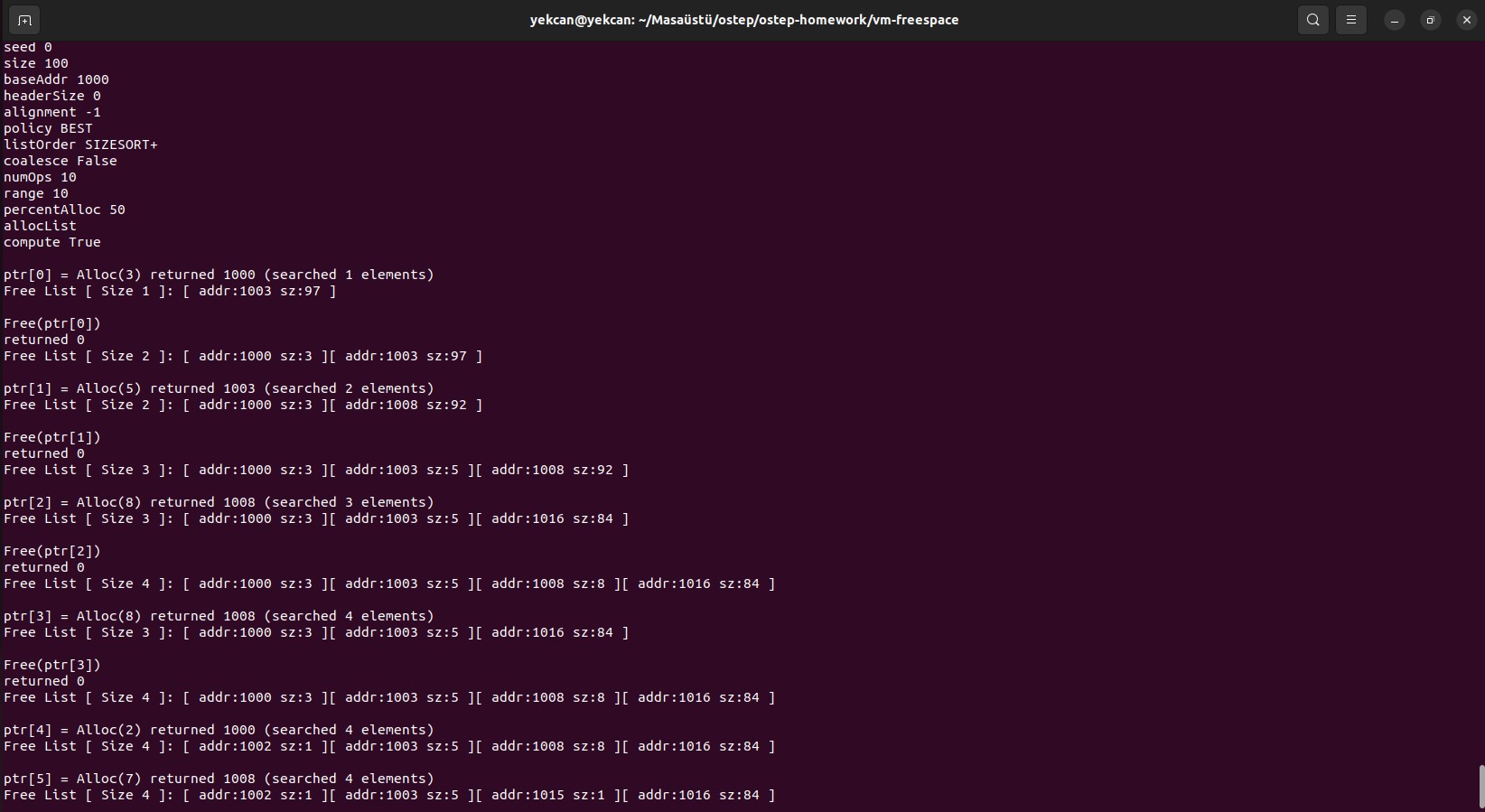
**“Adrese göre sıralama”**

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



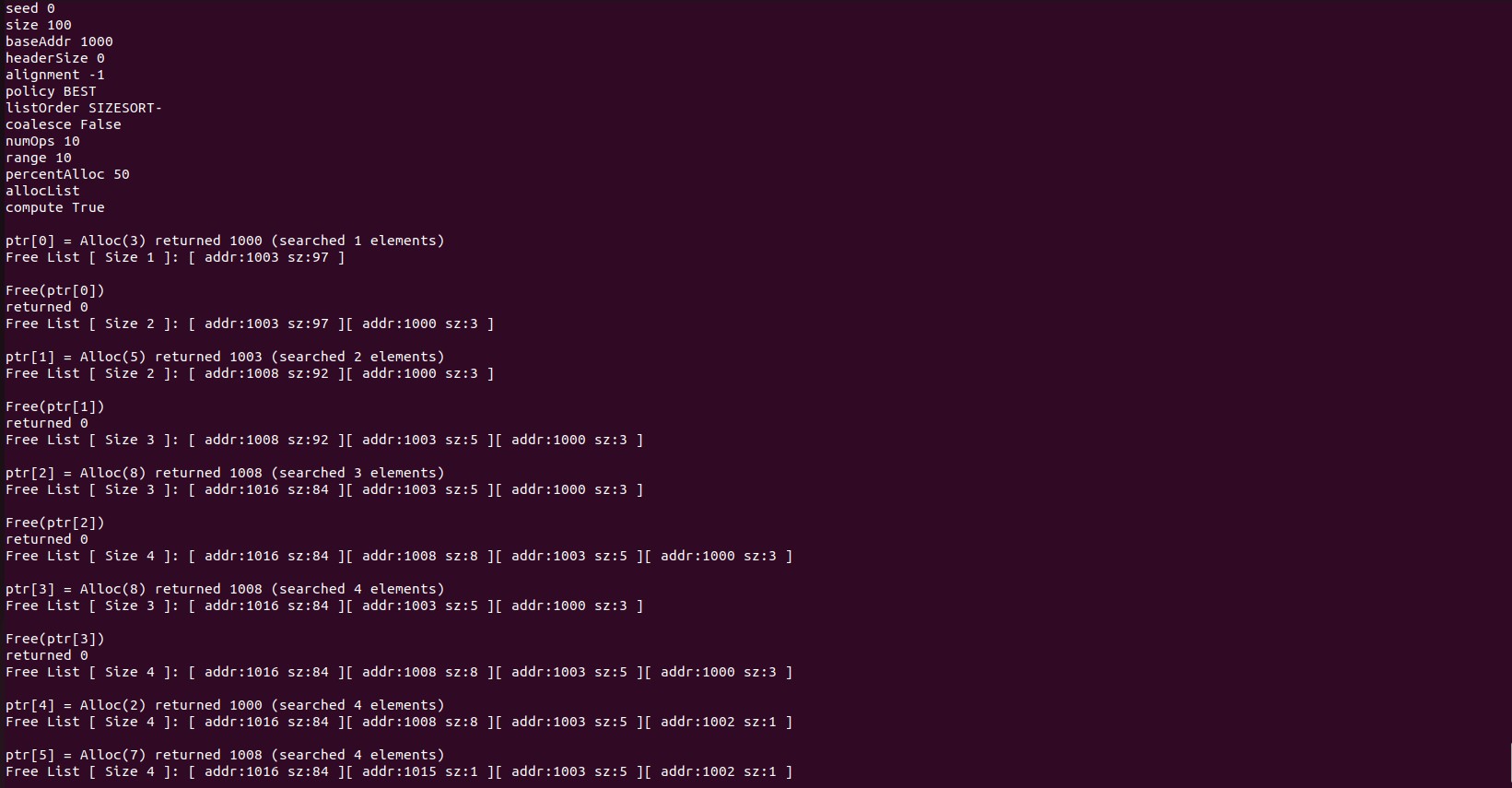
**“Artan büyüklük sırasına göre sıralama”**

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



**“Azalan büyüklük sırasına göre sıralama”**

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

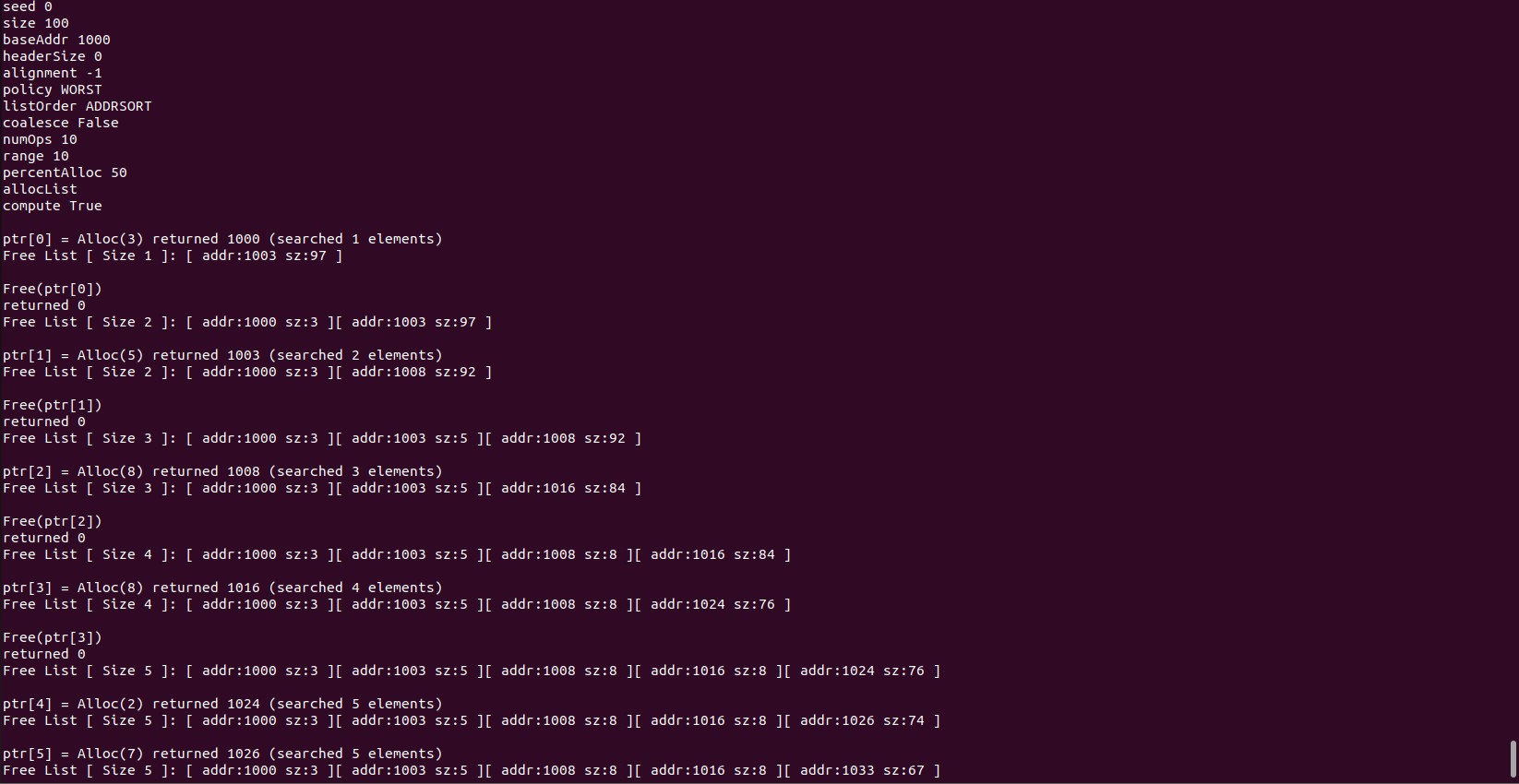


WORST Fit

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

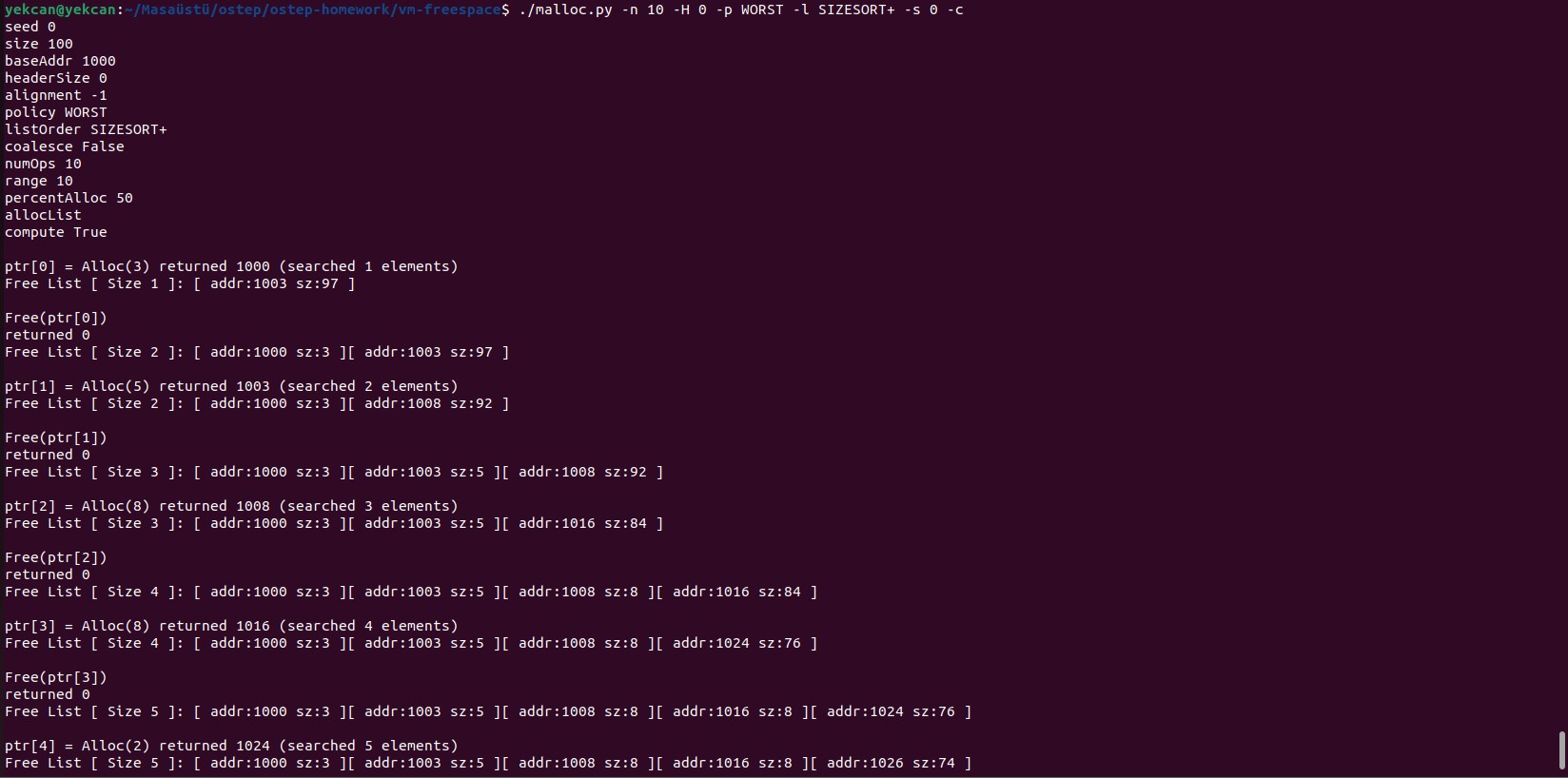
**“Adrese göre sıralama”**

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



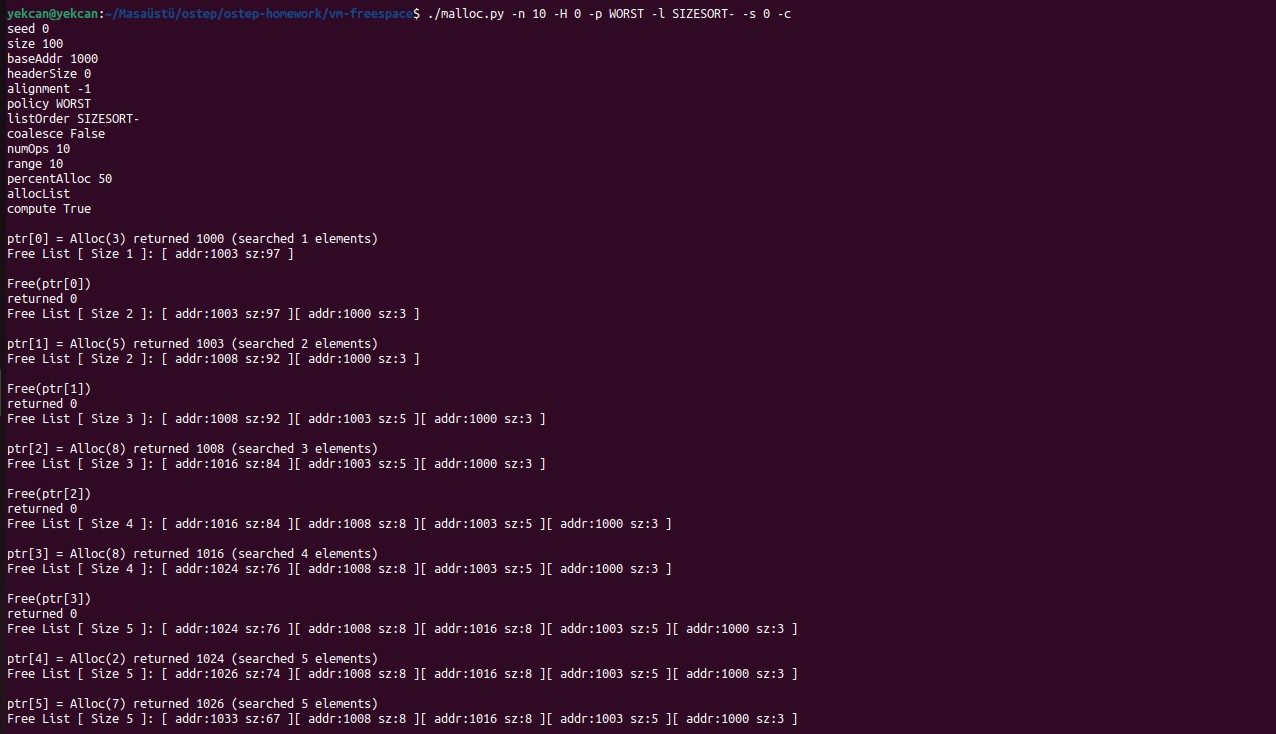
**“Artan büyüklük sırasına göre sıralama”**

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



**“Azalan büyüklük sırasına göre sıralama”**

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



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.

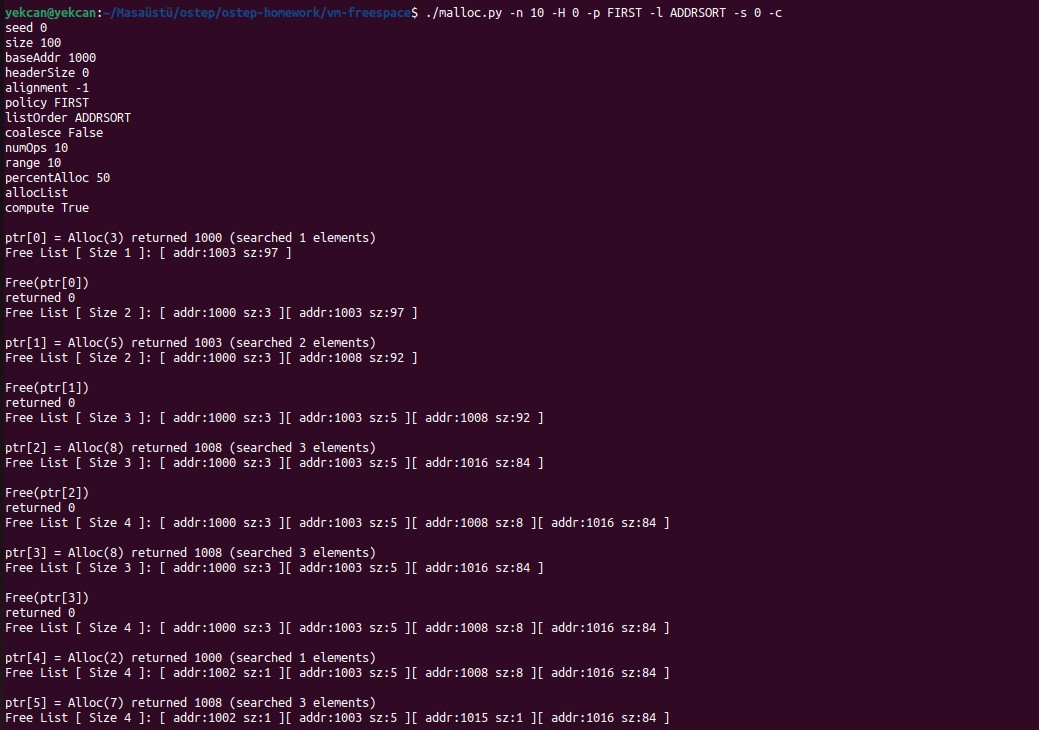
Tahsis boyutunu görmek ve rastgele karşılaştırmak için adres sırasına göre sıralayın.

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

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

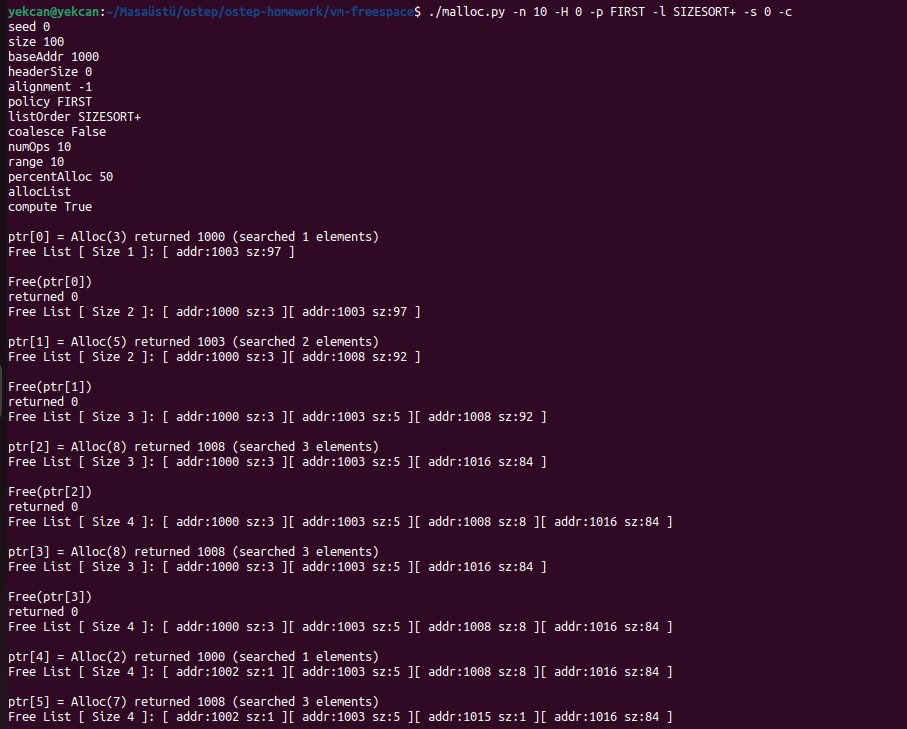
**“Adrese göre sıralama”**

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



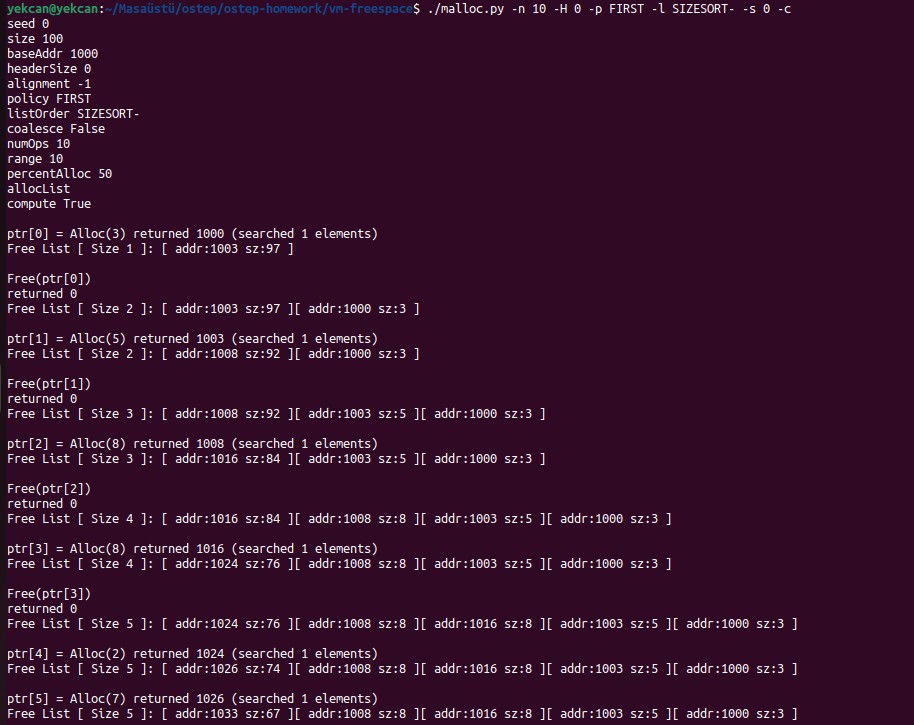
**“Artan büyüklük sırasına göre sıralama”**

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



**“Azalan büyüklük sırasına göre sıralama”**

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



1. **Serbest bir listenin birleştirilmesi oldukça önemli olabilir. Rastgele ayırma sayısını artırın (örneğin -n 1000'e). Zaman içinde daha büyük ayırma**

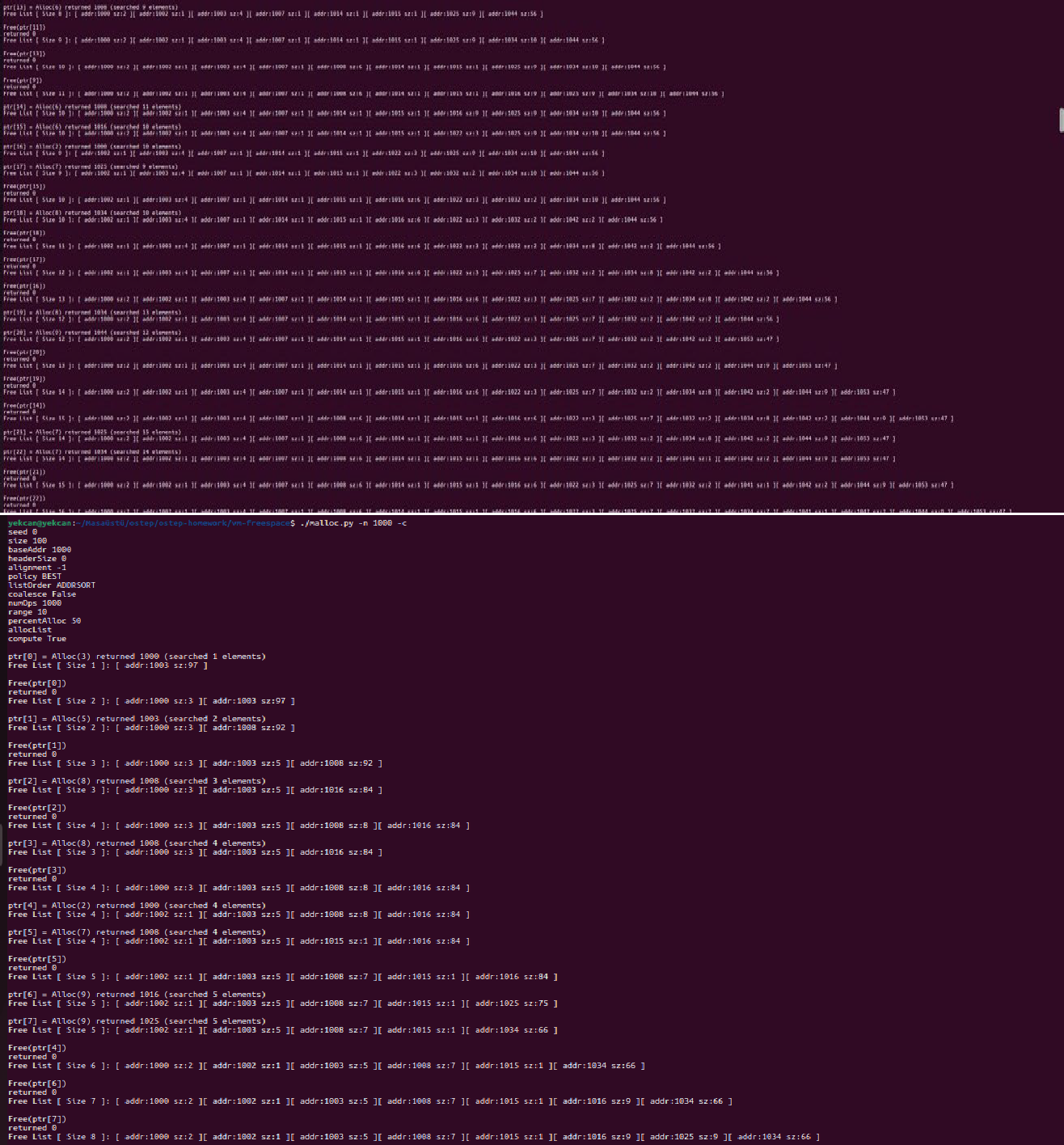
**isteklerine ne olur? Birleştirme ile ve birleştirmeden (yani -C bayrağı olmadan ve -C bayrağı ile) çalıştırın. Sonuçlarda ne gibi farklılıklar görüyorsunuz? Her durumda boş liste zaman içinde ne kadar büyür? Bu durumda listenin sıralaması önemli mi?**

**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 geçişi verimsiz hale getirir.

SONUÇLAR:

**SAYFALARCA SONUÇ ÇIKTIĞI İÇİN HEPSİNİ KOYAMIYORUM**.



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

**Peki ya yüzde 0'a yaklaştıkça?**

**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.

1. **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.