İşletim Sistemleri Dersi 2.Ödev Raporu

Memory Allocation Uygulaması

M.Yasin SAĞLAM

15011804

Ocak 2018

İşletim Sistemleri Dersi 2.Ödev Raporu

Memory Allocation Uygulaması

Giriş

Bu uygulamada C programlama dili kullanılarak Memory Allocation uygulaması olarak malloc, realloc ve free fonksiyonlarının implementasyonun gerçekleştirilmesi beklenmektedir. Öncelikle ödevin tanımında verildiği gibi Memory de bulunacak olan blockların header bilgileri bir struct içerisinde tutularak doubly linkli liste kullanılarak blockların Memory 'e find_free_block fonksiyonu ile first fit stratejisine göre yerleştirilmesi, malloc ile blockların genişletilmesi yada küçültülmesi realloc, ilgili bloğun memory de serbest bırakılması işlemi ise free fonksiyonları ile thread sade olarak gerçeklenmiştir.

İmplementasyon Detayları ve Analizi

Header bilgilerini tutan struct resimde görüldüğü ve açıklandığı gibi tanımlanmıştır.

```
//block yapisi
struct block{
    size_t size; //data kisminin boyutu
    int is_free; //blok bos mu, kullanimda mi
    struct block *next; //sonraki blogun adresi
    struct block *prev; //onceki blogun adresi
};
```

Daha sonrasında ise memory erişiminin thread safe olması için aşağıdaki resimde görüldüğü üzere global bir lock tanımlanmıştır.

```
pthread_mutex_t lock; //thread safe olmasi icin 1 adet global lock kullanmak yeterlidir
```

Linki liste yapısının tutulması içinde Linki listenin ilk elemanının adresini tutan bir head ve son elemanının adresini tutan bir tail aşağıdaki resimdeki gibi tanımlanmıştır.

```
struct block *head,*tail; //linkli listenin basi ve sonu
```

Sonrasında ise bize ilk boş olan uygun bloğun adresini döndürecek şekilde head den başlayarak arama yapan fonksiyon resimde görüldüğü ve açıklandığı üzere implemente edilmiştir.

```
//bos olan ilk blogun adresini donduren fonksiyon
//first fit allocation yapilmistir
struct block* find_free_block(size_t size){
    struct block *iter=head; //headi kaybetmemek icin itere atar gezeriz
    while (iter){ //daha once allocation yapilmissa
        if(iter->is_free && iter->size>=size) //eger bossa ve boyut olarak yeterliyse
            return iter; //blogu dondur
        iter=iter->next;// sart uygun degilse ilerle
    }
    return NULL; //uygun bos yer yok heapte yeni alan acilacak bu yuzden null dondur
}
```

Bu fonksiyon mm_alloc adlı allocation fonksiyonunda kullanılmak üzere tasarlanmıştır.

Bellekte uygun boş bir block bulunursa ilgili bloğa yerleşecek bulunamazsa yani linkli listenin sonuna gelinmişse ve hala uygun yer yoksa hepten sbrk() fonksiyonu kullanılarak yer ayrılacaktır ve data bloğu memset fonksiyonu ile 0 ile doldurulacaktır. Bu da mm_alloc fonksiyonunda aşağıda görüldüğü ve yorum satırlarında açıklandığı gibi thread safe olacak şekilde gerçekleştirilmiştir.

```
void *mm_malloc(size_t size) {
    struct block* here; //mevcut yeri tutan pointer
    size_t total_size;
    void* area; //bos olan alan adresi void ptr tipinde
    if(!size){
        printf("\nSize error Size must be greater than 0");
        return NULL;
    }
    //allocation baslasin
    pthread_mutex_lock(&lock);//thread safe olmasi icin lock aktif ediliyor
    here=find_free_block(size); //istenilen alan kadar ilk bos olan block araniyor
    if(here){ //eger bulunduysa
        here->is_free=0; //dolu yapiliyor
        pthread_mutex_unlock(&lock);//cikmadan once lock kalkiyor
        return (void*)(here+1); //header bilgisini ezmemek icin 1 sonraki adres yani datanin yerlesecegi adres donduruluyor
    }
    //eger null donduyse memory genisletilecek
    total_size=BLOCK_SIZE+size; //toplam boyut struct boyutu+istenen alan kadar
    area=sbrk(total_size); //heapi genislet
    if(area==(void*)-1){ //eger genisleme olmamissa donen degeri kontrol et
        printf("\nHeap Yetersiz"); //mesaj yazdir
        pthread_mutex_unlock(&lock);//cikmadan once lock kalkiyor
        return NULL; //NULL dondur
    }
    //basarili bir sekilde allocate edilmisse
    here=area; //mevcut yeri tutan pointer adresini heap area adresi yap
    here->is_free=0; //block dolu
    here->size=size; //block data boyutunu ata
    memset((void*)(here+1),0,size); //data blogu soruda istenildigi gibi sifirla dolduruluyor
    here->next=NULL; //sonrasinda bir blok gelmemistir sbrk ile heapi genislettik cunku
    if(!head){ //eger ilk kez yapiyorsak head degisir linli liste mantigi
        head=here;
    }
    if(!tail){ //tail de ayni sekilde
        tail=here;
    }
    else{ //eger oncesinde bir block varsa listenin sonuna ekleriz
        tail->next=here; //onceki blogun nexti olacak
        here->prev=tail; //ekledigimiz de onceki blogu tutacak
        tail=here; //son eklenen blok tail yapiliyor
    }
    pthread_mutex_unlock(&lock);//cikmadan once lock kalkiyor
    return (void*)(here+1); //head bilgisi atlanarak data yerlestirilecek blogun adresi donduruluyor
}
```

Realloc işlemi için mm_realoc adlı fonksiyon; ilgili blok küçültülecekse sadece header bilgisindeki size değişkeni set edilerek, aynı kalacaksa bloğun data kısmının adresi döndürülerek, genişletilecekse verilen boyutlara uygun bir boş blok linki listede bulunuyorsa oraya memcpy fonksiyonu ile kopyalanarak veya uygun bir block yoksa yine sbrk() kullanılarak heap ten block için yer Split edilerek eski değerlerin yeni yere memcpy fonksiyonu ile kopyalanarak eski bloğun free edilmesi şeklinde thread safe olarak gerçekleştirilmiştir.

Fonksiyon aşağıdaki resimde görüldüğü gibi implemente edilmiş ve açıklanmıştır.

```
//burada ilgili blogu tasiyabilecegimiz bir yer var mi ona bakacagiz
//yoksa malloc yapip eski blogu kopyaladiktan sonra eski blogu free yaparak bir cozum bulmus oluruz
void *mm_realloc(void *ptr, size_t size) {
    struct block* header;
    void* bigger;
    if(!ptr || size==0) //eger boyut sifirsa veya dizi yoksa malloc fonk gecis yapilir
        return malloc(size); //malloc zaten threadsafe burada gerek yok
    //burada threadsafe e gerek var
    pthread_mutex_lock(&lock); //lock koyulur memory erisilecek
    header=(struct block*)(ptr-1); //ilgili blogun header i cekiliyor
    if(header->size==size){//ayni boyuta realloc yapmak istiyorsa degisiklige gerek yok
        pthread_mutex_unlock(&lock);
        return ptr;
    }
    else if(header->size>size){//dizi kucultulmek isteniyorsa
        header->size=size; //header daki size degistirilir
        pthread_mutex_unlock(&lock);//lock kalkar
        return ptr;
    }
    else if(header->size<size){//dizi(data blogu) buyutulmek isteniyorsa
        pthread_mutex_unlock(&lock);//lock kalkar malloc calissin diye
        bigger=malloc(size); //daha buyuk bir yer ayrilir
        pthread_mutex_lock(&lock);//lock tekrar koyulur
        if(bigger){//eger basarili olmussa
            //yeni ayrilan alana eski alandaki bilgiler kopyalanir
            memcpy(bigger,ptr,header->size);//eski alandaki boyut kadar data blogu yeni data bloguna kopyalanir
            pthread_mutex_unlock(&lock);//lock kalkar free calissin diye
            free(ptr);//eski block free yapilir
            return bigger; //daha buyuk data blogu adresi dondurulur
        }
    }
    pthread_mutex_unlock(&lock);//lock kalkar cikilir
    return NULL;
}
```

Free işlemi için eğer free edilecek blok; heap in en sonunda yerleşmiş bulunan blok ise sbrk(0) ile mevcut konum alınıp en sonda chunk bir data bölümü kalmışsa o da hesaplanarak yine sbrk() komutuyla heap küçültülerek, değil ise ilgili bloğun header kısmında tutulan is_free değişkeni boş olacak şekilde set edilerek mm_free fonksiyonu aşağıdaki resimde görüldüğü ve açıklandığı gibi thread safe olacak şekilde implemente edilmiştir.

```
void mm_free(void *ptr) {
    struct block *header;
    void *end_heap;//heapin neresindeyiz en son dolu oldugu yer neresi
    size_t chunk;//saçma anlamsiz boslugu tutar heap in son kaldigi yer ile son blogun data blogundan sonraki kisim
    if(!ptr)  //eger verilen adreste bisey yoksa
        return; //cikis
    //varsa free yapacagiz
    pthread_mutex_lock(&lock);//threadsafe icin lock aktif olur
    //header bilgisinin adresi ataniyor
    header=(struct block*)(ptr-1); //data adresinin 1 byte gerisinde ilgili header bulunuyor
    end_heap=sbrk(0);//sbrk(0) bize heapin mevcut kalinan en son yerini verecektir.
    //simdi ilgili block en sonda ise heapi kucultmeliyiz degilse blogun isfree degiskenini 1 yapmaliyiz
    //boylelikle tekrardan oraya baska bilgi yazilabilir.
    //blogun degerlerinin en sona yerlestigini anlamak icin
    //baslangic arti data ile blogun en son nerede bittigini buluruz
    if(header->next==NULL){//eger o blogun sonrasi null ise son block demektir.
        if(head==tail){//eger 1 tane block kalmissa
            head=tail=NULL; //linkli liste baslangic durumunu alir
        }
        else{//son blogun adresi linkli listeden cikarilir
            tail=tail->prev;//yani tail bir onceki eleman olur ve nexti null olur
            tail->next=NULL;
        }
        //sonrasinda ise sbrk ile heap kucultulur
        //kucultme miktari ise elimizdeki artik alan+header boyutu+header da tutulan block size kadar olur
        chunk=end_heap-(ptr+header->size); //artik alan demektir
        sbrk(0- BLOCK_SIZE-header->size-chunk);
        //locku kaldirir ve return ederiz
        pthread_mutex_unlock(&lock);
        return;
    }
    //eger free yapilmak istenen block son block degilse
    //sadece header yapisindaki isfree degiskenini 1 yapariz
    header->is_free=1;
    pthread_mutex_unlock(&lock);//locku kaldiririz
}
```

SONUÇ

Bu uygulamada C programlama dilinde sürekli kullandığımız memory management (malloc,realloc,free). fonksiyonlarının seçilen first fit stratejisine göre ve thread safe olacak şekilde memory' e erişirken proseslerin uygun bir şekilde(race condition, critical section problemi vs kaçınarak) erişmeleri için bir global lock mekanizması kurularak, iki yönlü linki liste(doubly linked list) veri yapısı kullanılarak implementasyonu gerçekleştirilmiştir ve bu fonksiyonların arkaplanda memory' i nasıl yönettiklerine dair bilgi ve tecrübe edinilmiştir.