Bellek Yönetimi Projesi

Boş bellek bölgeleri bağlı listede (linked list) tutulacak. Bu listede her düğüm (node) bir boş bloğun başlangıç adresini ve bitiş adresini tutacak. Ayrıca her bir düğüm kendinden bir sonraki düğümü gösteriyor olacak. Dolu olan (yani kullanımda olan) bellek bölgeleri bağlı listede yer almayacak; bağlı liste sadece boş bellek alanlarını temsil eden düğümlerden oluşacak.

```
struct dugum {
      int bas;
      int son;
      struct dugum * next;
};

void bellekAlAdresli(struct dugum** root, int startAddr, int size);
void bellekAlAdressiz(struct dugum** root, int size);
void bellekIadeEt(struct dugum** root, int startAddr, int size);
```

Bu 3 fonksiyonu gerçeklemeni (implement etmeni) istiyorum. Ekteki örnek çıktıyı incelersen fonksiyonların çalışma mantığını görebilirsin.

`bellekAlAdresli` fonksiyonu parametre olarak verilen **`startAddr`** adresinden itibaren **`"size"`** boyutunda bir alanın tümü boş ise bu alanı verecek ve bağlı listeyi güncelleyecek. Bir bölümü boş bir bölümü dolu olan alan için hiçbir şey yapmayacak.

`bellekAlAdressiz` fonksiyonu parametre olarak verilen **`"size"`** boyutunda bir boş alanı verecek ve bağlı listeyi güncelleyecek. Bu fonksiyonu first-fit değil best-fit algoritmasına göre tasarlamanı istiyorum. Yani bellekte fragmantasyon az olsun diye mümkün oldukça büyük düğümleri parçalamadan, istenen alana en yakın boyuttaki bölgeden tahsis yapılacak. Örneğin 2 kilobyte yer isteniyorsa bağlı liste gezilecek, 2 kilobayta büyük eşit alanı olan bölgelerin en küçüğünden tahsis yapılacak.

`bellekIadeEt` fonksiyonu parametre olarak verilen **`startAddr`** adresinden itibaren parametre olarak verilen **`"size"`** boyutunda alanın tümü dolu ise bu alanın iadesine izin verecek ve bağlı listeyi güncelleyecek. Bir bölümü dolu bir bölümü boş olan alan için hiçbir şey yapmayacak.

Ekteki örnek çıktıdan daha iyi anlayabileceğini düşünüyorum. Başlangıçta bütün bellek kullanımda diye düşünelim. Yani bağlı liste başlangıçta hiç düğüm içermiyor.

```
bellekIadeEt(&pList, 900 * kb, 5 * kb)
----------------------------------------------------------------
900K-905K
```

Bu komutla 900 Kilobayt - 905 Kilobayt arası free edildikten sonra ilk düğüm oluşuyor.

Daha sonra 26 KB adresinden itibaren 3 KB'lik alan free ediliyor. Bu tahsis sonucu bir düğüm daha oluşuyor ve bağlı liste aşağıdaki hali alıyor:

```
bellekIadeEt(&pList, 26 * kb, 3 * kb)
------------------------------------------------------------------
26K-29K 900K-905K
```

Daha sonra son iade edilen alan tekrar tahsis ediliyor. Böylece bağlı liste önceki haline dönüyor.

```
bellekAlAdresli(root, 26 * kb, 3 * kb)
------------------------------------------------------------------
900K-905K
```

Diğer adımları ekteki dosyadan incelersen olayı tamamen anlayabilirsin

Memory Management Project

The goal of this task is to implement three functions in the context of managing memory allocation and deallocation using a linked list. The linked list will maintain regions of free memory blocks. Each node in the list will store the starting and ending addresses of an empty memory block. Additionally, each node will point to the next node. The linked list will only represent empty memory areas; occupied (used) memory regions will not be included.

```
struct node {
    int start;
    int end;
    struct node* next;
};
```

There are three functions to be implemented:

```
void allocateMemoryAddressed(struct node** root, int startAddr, int size);
void allocateMemoryAddressless(struct node** root, int size);
void deallocateMemory(struct node** root, int startAddr, int size);
```

`allocateMemoryAddressed:` This function allocates a block of memory starting from the **`startAddr`** address, with a specified **`"size"`**. It updates the linked list accordingly. It will not do anything if the memory block is partially or fully occupied.

`allocateMemoryAddressless:` This function allocates a block of memory with the given size using the best-fit algorithm. Do not use the first-fit algorithm. It aims to minimize fragmentation by allocating from the closest available block without breaking larger blocks. The linked list will be updated to reflect the allocation.

`deallocateMemory:` This function deallocates a block of memory starting from the **`startAddr`** address, with a specified **`"size"`**. It updates the linked list to mark the deallocated region as free. It will not do anything if the memory block is partially or fully unoccupied.

The provided sample output demonstrates how the functions work:

```
deallocateMemory(&pList, 900 * kb, 5 * kb)
----------------------------------------------------------------
900K-905K
```

After freeing the memory from 900 KB to 905 KB, the first node is created.

Later, a 3 KB block is deallocated from the address 26 KB. This results in another node creation, and the linked list gets updated:

```
deallocateMemory(&pList, 26 * kb, 3 * kb)
----------------------------------------------------------------
26K-29K 900K-905K
```

Subsequently, the last deallocated block is allocated again, and the linked list reverts to its previous state:

```
allocateMemoryAddressed(root, 26 * kb, 3 * kb)
-----------------------------------------------------------------
900K-905K
```

For a complete understanding of the process, you can refer to the attached file detailing the additional steps.

The main concepts here are managing memory blocks using a linked list and implementing allocation and deallocation strategies. The best-fit algorithm is used to minimize fragmentation when allocating memory. The sample output provides clear examples of how the functions affect the linked list and memory allocation.