Memory Management (DXE) Code Analysis

原创 Pedro Posted on 2017-01-02 14:20:40 Read 3k Collection 6 Likes 5

Category Column: UEFI-MemoryManagement Article Tags: Memory allocation Code Analysis

摘要 This article analyzes the CoreInitializeMemoryServices function in the UEFI DXE stage in detail, introducing the initialization process of the memory pool (Pool) and the role of the CoreAddMemoryDescriptor function. The latter prepares for subsequent memory allocation services and involves complex memory descriptor management.

The summary is generated in C Know , supported by DeepSeek-R1 full version, go to experience>

Regarding code analysis, this time we will start with CoreInitializeMemoryServces (DxeMain.c).
This function is the first function related to initializing memory in the DXE stage. Let's sort out this function first. It mainly does the following three things:

1 Initialize the data structure related to Pool memory allocation
2 Find the memory address that can accommodate DxeCore by searching Hob
3 Prepare for the next memory allocation services through the CoreAddMemoryDescriptor function

Here we mainly introduce the contents of points 1 and 3.

CoreInitializePool

AI generated projects 登录复制

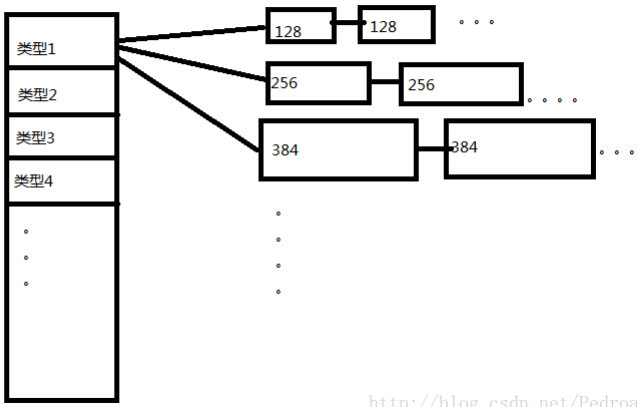
```
UINTN  Type;
UINTN  Index;

for (Type=0; Type < EfiMaxMemoryType; Type++) {
  mPoolHead[Type].Signature  = 0;
  mPoolHead[Type].Used       = 0;
  mPoolHead[Type].MemoryType = (EFI_MEMORY_TYPE) Type;
  for (Index=0; Index < MAX_POOL_LIST; Index++) {
    InitializeListHead (&mPoolHead[Type].FreeList[Index]);
  }
}

```

收起 ^

The Pool initialization function is very simple. Our Pool will look like the following depending on the memory type and size.



Pool is first divided by type, and then each type is divided by the size of the block. This function mainly initializes the linked list of this structure.

The CoreAddMemoryDescriptor
function is relatively complex and very important. If you have read the previous article, then the analysis of this function will be much simpler. This function is actually like this. For example, the administrator who manages memory Allocate has nothing in his hands at the beginning, and then CoreAddMemoryDescriptor is used to allocate a memory block of a certain type and size to this administrator. Calling CoreAddMemoryDescriptor in CoreInitializeMemoryServices is for the purpose of allocating memory next, but as we mentioned before, the system has to apply to GCD to allocate memory to each administrator, but the first time this time the GCD management department has not been established, the memory allocation administrator will use it first, and wait for GCD to be initialized, and then go to register, that is, get on the bus first and buy the ticket later. Because it is complicated enough, let us first briefly introduce its specific functions:

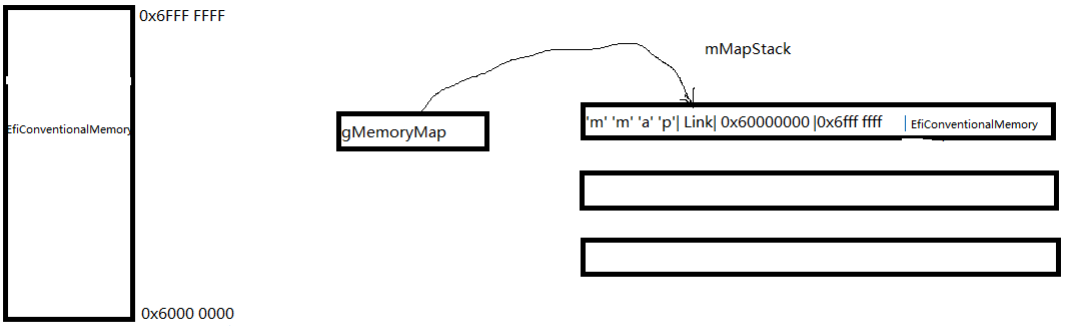
- Initialize gMemMap information and register the added memory block information mainly through CoreAddRange and CoreFreeMemoryMapStack two functions
- The bunch of for loops in the middle are mainly about the introduction in the previous articles. In order to keep the reserved type memory unchanged and reduce memory fragmentation when S4 resumes, we will draw memory blocks one by one in the code, called various types of bin files. We can see that in a bunch of for loops, the bins of various types of memory are first applied for and then released. The release after application has two purposes: 1. Record the starting address and size of each type of memory bin. The information is recorded in mMemoryTypeStatistics. When the relevant type of memory is to be allocated later, first look for the address recorded in the mMemoryTypeStatistics information and allocate it in that address range. If the recorded range is not enough, then go to another address range to allocate this type. 2. Test whether the memory currently allocated by the system to the memory manager has enough space to accommodate the memory type bin files reserved in the code.

This is roughly what we have done above. Regarding the first point, let's look at the following code.

```
544   CoreAcquireMemoryLock ();
545   End = Start + LShiftU64 (NumberOfPages, EFI_PAGE_SHIFT) - 1;
546   CoreAddRange (Type, Start, End, Attribute);
547   CoreFreeMemoryMapStack ();
548   CoreReleaseMemoryLock ();
```

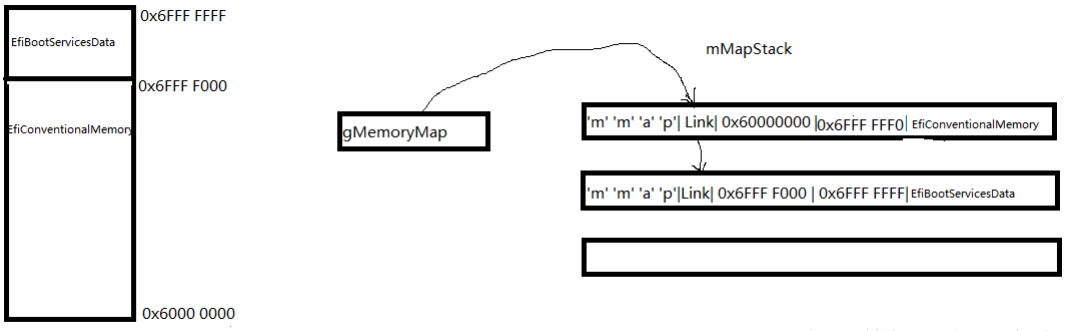
CoreAddRange:

Mainly adds an Entry, this Entry has various attribute information of this memory range, and then links to gMemMap through a linked list. In the while loop, two Entries are merged. If the memory blocks represented by these two Entries are continuous and have the same attributes, they can be merged. Because the Entry that records the memory block also requires space, the approach here is to first store it in the DxeCore space and name it mMapStack. Later we will see that the code will allocate another space specifically for storing all entries. mMapStack also has a parameter called mMapDepth, which manages the depth of mMapStack. Because there will be another CoreAddRange in the function that releases mMapStack, that is, a recursive call, there will be a parameter mMapDepth to record how many entries currently need to be released in mMapStack. After execution, it will be as shown below:

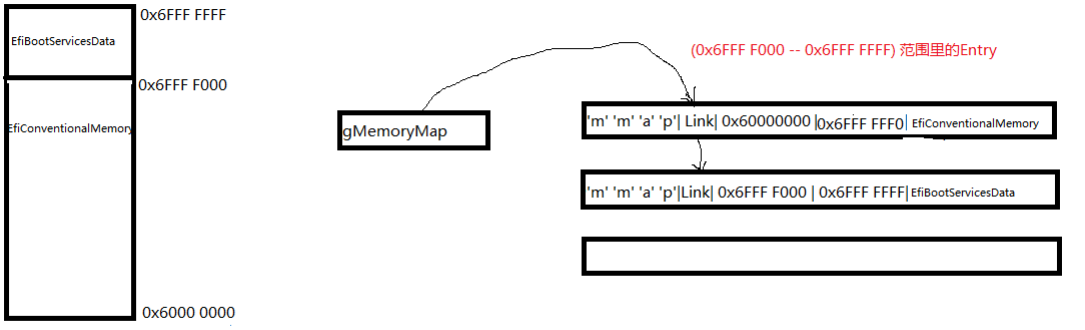


CoreFreeMemoryMapStack:

This function releases the Entry of mMapStack, and then allocates a large space for recording the Entry and manages it with the linked list mFreeMemoryMapEntry. It also copies the Entry content in mMapStack to the Entry of mFreeMemoryMapEntry, and then links the Entry in mFreeMemoryMapEntry to gMemoryMap. Because recursive calls will occur, a lock mFreeMapStack is set in this function. Then a while loop follows to release all the Entry according to the number of recursive layers of mMapStack. Regarding this function, let's first look at the call to the AllocateMemoryMapEntry function. The AllocateMemoryMapEntry function allocates a space in the interval block (0x6000 0000 — 0x6FFF FFFF) in the above figure to record the Entry information. AllocateMemoryMapEntry is executed, the picture is as follows:



The next step is to copy the content of the Entry in mMapStack to the Entry in (0x6FFF F000 - 0x6FFF FFFF). And point the gMemoryMap linked list to the Entry in (0x6FFF F000 - 0x6FFF FFFF). As shown below:



Next is a bunch of For loop code, which has been briefly mentioned above. The detailed analysis will be introduced later when analyzing AllocatePool and AllocatePage.