Memory Management (DXE) Example Analysis

原创 Pedro Posted on 2017-01-11 11:28:37 Read 5.2k Collection 2 Likes 1 Copyright CC 4.0 BY-SA
Category Column: UEFI-MemoryManagement Article Tags: uefi

摘要 This article explores the memory management mechanism in the UEFI environment in depth, focusing on the different types of AllocatePages functions and their internal implementation principles, including the specific functions of FindFreePages and CoreConvertPages.
The summary is generated in C Know , supported by DeepSeek-R1 full version, go to experience>

This article uses an example to analyze AllocatePages() to consolidate and deepen the previous knowledge points and connect them together.
First, let's take a look at the definition of AllocatePages under UEFI .

```
typedef
EFI_STATUS
(EFIAPI *EFI_ALLOCATE_PAGES) (
IN EFI_ALLOCATE_TYPE Type,
IN EFI_MEMORY_TYPE MemoryType,
IN UINTN Pages,
IN OUT EFI_PHYSICAL_ADDRESS*Memory
);
```

Here we will briefly explain the first parameter Type.
EFI_ALLOCATE_TYPE is divided into 3 types:

- AllocateAnyPages
- AllocateMaxAddress
- AllocateAddress

AllocateAnyPages: When allocating memory, the memory manager will look for an address that can be allocated. The location of the allocated memory is specified by the memory manager, and the fourth parameter passed in will be ignored.

AllocateMaxAddress: This type tells the manager memory that I specify an address, and the address you allocate to me cannot exceed the address passed in by the fourth parameter

AllocateAddress: This type tells the memory manager that you want to give me an address, the location is specified by the fourth parameter. This type is used in some special cases because it is easy to cause memory fragmentation.

The AllocatePages function mainly calls CoreInternalAllocatePages to allocate memory. The following is a brief analysis of CoreInternalAllocatePages.

The CoreInternalAllocatePages body is divided into the three types just introduced as follows:

```
  MaxAddress = MAX_ADDRESS;
  if (Type == AllocateMaxAddress)
    MaxAddress = *Memory;

  if (Type != AllocateAddress) {
    Start = FindFreePages (MaxAddress, NumberOfPages, MemoryType, Alignment);
  }

  Status = CoreConvertPages (Start, NumberOfPages, MemoryType);
```

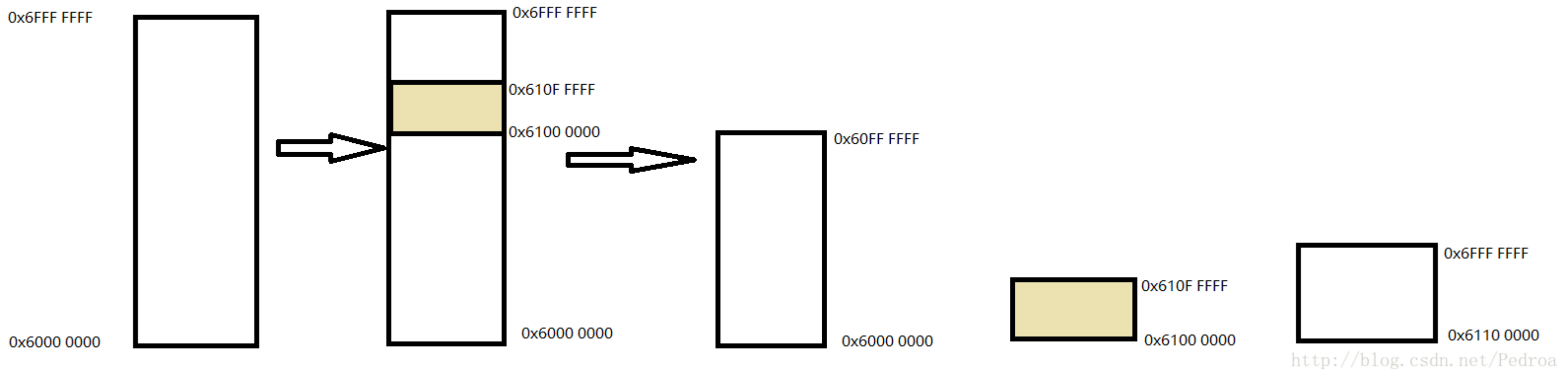
The above extracts CoreInternalAllocatePages and removes some other judgment information. The main thing is to understand the general idea first, and you can only study the details yourself.

The meaning of the FindFreePages
lines of code is that if the address of the allocated memory is not specified, the memory allocator will check if there is a block in its own warehouse that meets the memory size that the user wants to allocate. Of course, the search is conditional, that is, to see if there is a MaxAddress. If it is found, it will be taken out, otherwise forget it.

CoreConvertPages
is the function that starts the allocation

Well, before we analyze FindFreePages and CoreCovertPages in detail, let's take a look at the following diagram. Looking at the diagram will help us build a model, and there will be no obstacles when looking at the code.

For example, we now have a block of memory (0x6000 0000 - 0x6FFFF FFFF) in the memory manager warehouse, and now the application has requested a block of memory with a specified address of 0x6100 0000, a size of 1M, and a type of BootServicesData. The picture shows the changes in gMemoryMap.



This picture is probably what the CoreConvertPages function does.

Now let's analyze FindFreePages briefly. Remember the previous blog introduced that in order to ensure that some reserved addresses of memory cannot be changed when S4 resumes, UEFI adopts a simple and direct method, which is to pre-allocate bin files of different sizes according to types, and then when APP or driver allocates memory in the future, it will be allocated in bin first. Then FindFreePages does this.

```
FindFreePages:
  //
  // Attempt to find free pages in the preferred bin based on the requested memory type
  //
  if ((UINT32)NewType < EfiMaxMemoryType && MaxAddress >= mMemoryTypeStatistics[NewType].MaximumAddress) {
    Start = CoreFindFreePagesI (
              mMemoryTypeStatistics[NewType].MaximumAddress,
              mMemoryTypeStatistics[NewType].BaseAddress,
              NoPages,
              NewType,
              Alignment
              );
    if (Start != 0) {
      return Start;
    }
  }
```

Extract the fragments. First, when FreePage is found, it will go to the preferred bin to find it. Then the size and base of the pre-allocated preferred bin are stored in mMemoryTypeStatistics. When mMemoryTypeStatistics is initialized, it is in the for loop mentioned before in CoreAddMemoryDescriptor. If it is not found in the preferred bin, there is a default bin to allocate. If it still fails, it can only be allocated somewhere else. In this way, as mentioned before, the pre-defined bin is not enough to accommodate the actual memory requested by the driver or App. It is necessary to restart in bds and then expand the pre-defined preferred bin size to the size of the actual memory used last time.

What CoreConvertPages does is as shown in the picture above. First, it will find its own free memory block. If it contains the memory block to be allocated, it will be found. Then it will reorganize the memory block. This code is actually very core, but it is introduced in a blog. If I read other people's blogs and post too much code, I can't read it. debug, to basically trace each section of code. It took more than a week. Many of the places, such as locks, mutually exclusive flags, recursive function calls, and the use of mMapStack, can only be understood by tracing them once. I won't write it down, there are too many.