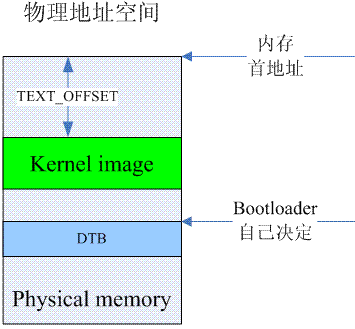
基于ARM64的内存初始化

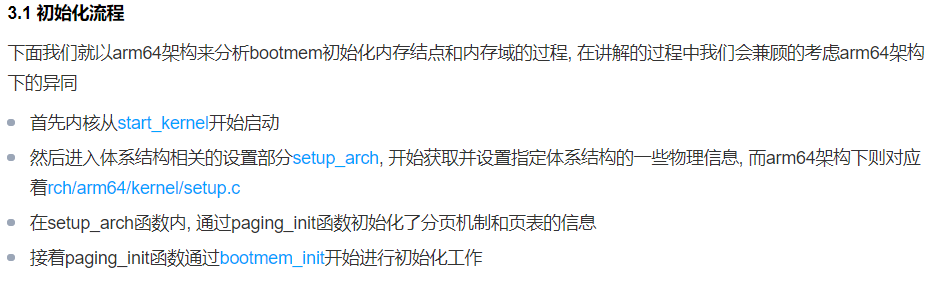
启动之前：

在详细描述linux kernel对内存的初始化过程之前，我们必须首先了解kernel在执行第一条语句之前所面临的处境。这时候的内存状况可以参考下图：



bootloader有自己的方法来了解系统中memory的布局，然后，它会将绿色的kernel image和蓝色dtb image copy到了指定的内存位置上。kernel image最好是位于main memory起始地址偏移TEXT\_OFFSET的位置，当然，TEXT\_OFFSET需要和kernel协商好。kernel image是否一定位于起始的main memory（memory address最低）呢？也不一定，但是对于kernel而言，低于kernel image的内存，kernel是不会纳入到自己的内存管理系统中的。对于dtb image的位置，linux并没有特别的要求。由于这时候MMU是turn off的，因此CPU只能看到物理地址空间。对于cache的要求也比较简单，只有一条：kernel image对应的cache必须clean to PoC，即系统中所有的observer在访问kernel image对应内存地址的时候是一致性的。

内核掌控内存系统控制权之后：



内存管理是操作系统资源管理的重点, 但是在操作系统初始化的初期, 操作系统只是获取到了内存的基本信息, 但是内存管理的数据结构都没有建立, 而我们这些数据结构创建的过程本身就是一个内存分配的过程, 那么就出现一个问题

我们还没有一个内存管理器去负责分配和回收内存, 而我们又不可能将所有的内存信息都静态创建并初始化, 那么我们怎么分配内存管理器所需要的内存呢? 现在我们进入了一个先有鸡还是先有蛋的怪圈, 这种问题的一般解决方法是, 我们先实现一个满足要求的但是可能效率不高的笨家伙(内存管理器), 用它来负责系统初始化初期的内存管理, 最重要的, 用它来初始化我们内存的数据结构, 直到我们真正的内存管理器被初始化完成并能投入使用, 我们将旧的内存管理器丢掉

即因此在系统启动过程期间, 内核使用了一个额外的简化形式的内存管理模块早期的引导内存分配器(boot memory allocator–bootmem分配器)或者memblock, 用于在启动阶段早期分配内存, 而在系统初始化完成后, 该分配器被内核抛弃, 然后初始化了一套新的更加完善的内存分配器.

2.1 引导内存分配器bootmem

在启动过程期间, 尽管内存管理尚未初始化, 但是内核仍然需要分配内存以创建各种数据结构, 早期的内核中负责初始化阶段的内存分配器称为引导内存分配器(boot memory allocator–bootmem分配器), 在耳熟能详的伙伴系统建立前内存都是利用分配器来分配的，伙伴系统框架建立起来后，bootmem会过度到伙伴系统. 显然, 对该内存分配器的需求集中于简单性方面,　而不是性能和通用性, 它仅用于初始化阶段. 因此内核开发者决定实现一个最先适配(first-first)分配器用于在启动阶段管理内存. 这是可能想到的最简单的方式.

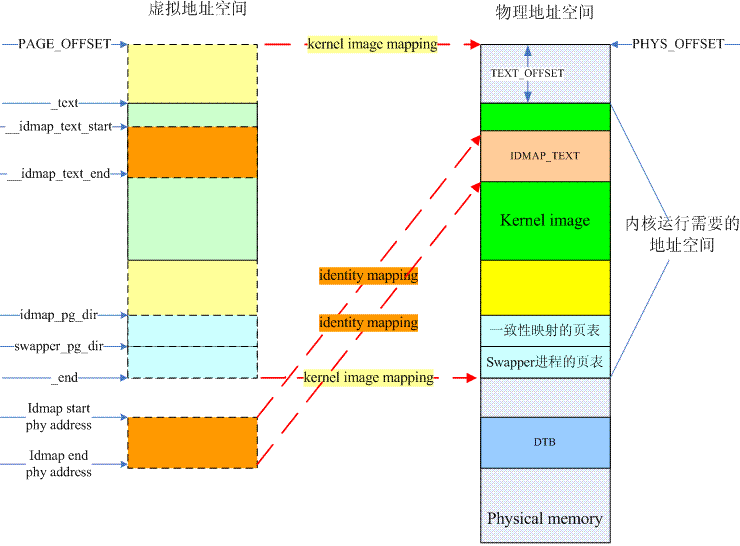
引导内存分配器(boot memory allocator–bootmem分配器 )基于最先适配(first-first)分配器的原理(这儿是很多系统的内存分配所使用的原理), 使用一个位图来管理页, 以位图代替原来的空闲链表结构来表示存储空间, 位图的比特位的数目与系统中物理内存页面数目相同. 若位图中某一位是1, 则标识该页面已经被分配(已用页), 否则表示未被占有(未用页).

在需要分配内存时, 分配器逐位的扫描位图, 直至找到一个能提供足够连续页的位置, 即所谓的最先最佳(first-best)或最先适配位置.该分配机制通过记录上一次分配的页面帧号(PFN)结束时的偏移量来实现分配大小小于一页的空间, 连续的小的空闲空间将被合并存储在一页上.

即使是初始化用的最先适配分配器也必须使用一些数据结构, 内核为系统中每一个结点都提供了一个struct bootmem\_data结构的实例, 用于bootmem的内存管理. 它含有引导内存分配器给结点分配内存时所需的信息. 当然, 这时候内存管理还没有初始化, 因而该结构所需的内存是无法动态分配的, 必须在编译时分配给内核.

一旦跳转到linux kernel执行，内核则完全掌控了内存系统的控制权，它需要做的事情首先就是要打开MMU，而为了打开MMU，必须要创建linux kernel正常运行需要的页表，这就是本节的主要内容。

在体系结构相关的汇编初始化阶段，我们会准备二段地址的页表：一段是identity mapping，其实就是把地址等于物理地址的那些虚拟地址mapping到物理地址上去，打开MMU相关的代码需要这样的mapping（别的CPU不知道，但是ARM ARCH强烈推荐这么做的）。第二段是kernel image mapping，内核代码的执行当然需要将kernel running需要的地址（kernel txt、rodata、data、bss等等）进行映射了。具体的映射情况可以参考下图：



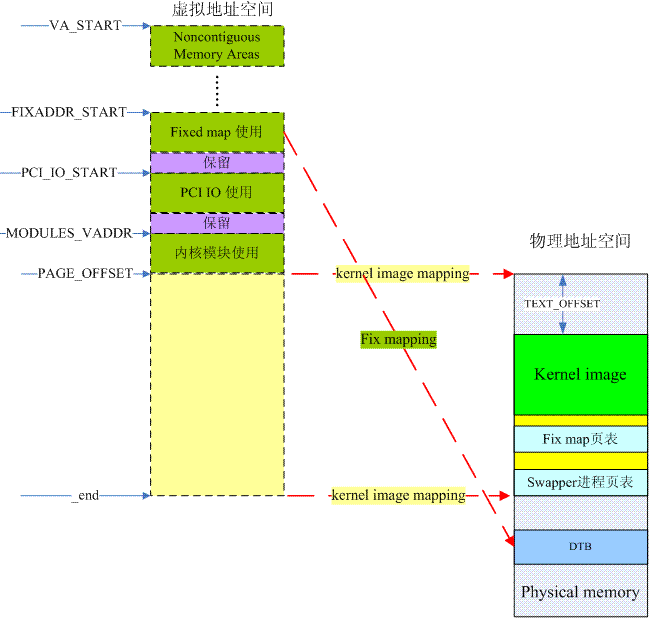
编译器感知的是kernel image的虚拟地址（左侧），在内核的链接脚本中定义了若干的符号，都是虚拟地址。但是在内核刚开始，没有打开MMU之前，这些代码实际上是运行在物理地址上的，因此，内核起始刚开始的汇编代码基本上是PIC的，首先需要定位到页表的位置，然后在页表中填入kernel image mapping和identity mapping的页表项。页表的起始位置比较好定（bss段之后），但是具体的size还是需要思考一下的。我们要选择一个合适的size，确保能够覆盖kernel image mapping和identity mapping的地址段，然后又不会太浪费。

swapper进程（内核空间）需要预留页表的size是和page table level相关，如果使用了section mapping，那么需要预留PGTABLE\_LEVELS - 1个page。如果不使用section mapping，那么需要预留PGTABLE\_LEVELS 个page。

一旦设定完了页表，那么打开MMU之后，kernel正式就会进入虚拟地址空间的世界

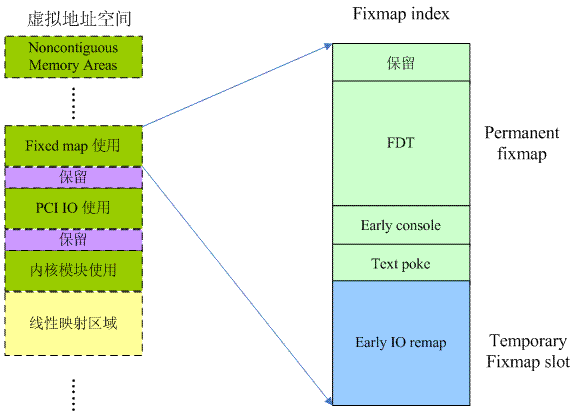
虽然可以通过kernel image mapping和identity mapping来窥探物理地址空间，但终究是管中窥豹，不了解全局，那么内核是如何了解对端的物理世界呢？答案就是DTB，但是问题来了，这时候，内核还没有为DTB这段内存创建映射，因此，打开MMU之后的kernel还不能直接访问，需要先创建dtb mapping，而要创建address mapping，就需要分配页表内存，而这时候，还没有了解内存布局，内存管理模块还没有初始化，如何来分配内存呢？

下面这张图片给出了解决方案：



整个虚拟地址空间那么大，可以被平均分成两半，上半部分的虚拟地址空间主要各种特定的功能，而下半部分主要用于物理内存的直接映射。对于DTB而言，我们借用了fixed-mapped address这个概念。fixed map是被linux kernel用来解决一类问题的机制，这类问题的共同特点是：（1）在很早期的阶段需要进行地址映射，而此时，由于内存管理模块还没有完成初始化，不能动态分配内存，也就是无法动态分配创建映射需要的页表内存空间。（2）物理地址是固定的，或者是在运行时就可以确定的。对于这类问题，内核定义了一段固定映射的虚拟地址，让使用fix map机制的各个模块可以在系统启动的早期就可以创建地址映射，当然，这种机制不是那么灵活，因为虚拟地址都是编译时固定分配的。

除了DTB，在启动阶段，还有其他的模块也想要创建地址映射，当然，对于这些需求，内核统一采用了fixmap的机制来应对，fixmap的具体信息如下图所示：



从上面这个图片可以看出fix-mapped虚拟地址分成两段，一段是permanent fix map，一段是temporary fixmap。所谓permanent表示映射关系永远都是存在的，例如FDT区域，一旦完成地址映射，内核可以访问DTB之后，这个映射关系一直都是存在的。而temporary fixmap则不然，一般而言，某个模块使用了这部分的虚拟地址之后，需要尽快释放这段虚拟地址，以便给其他模块使用。

完成DTB的映射之后，内核可以访问这一段的内存了，通过解析DTB中的内容，内核可以勾勒出整个内存布局的情况，为后续内存管理初始化奠定基础。收集内存布局的信息主要来自下面几条途径：

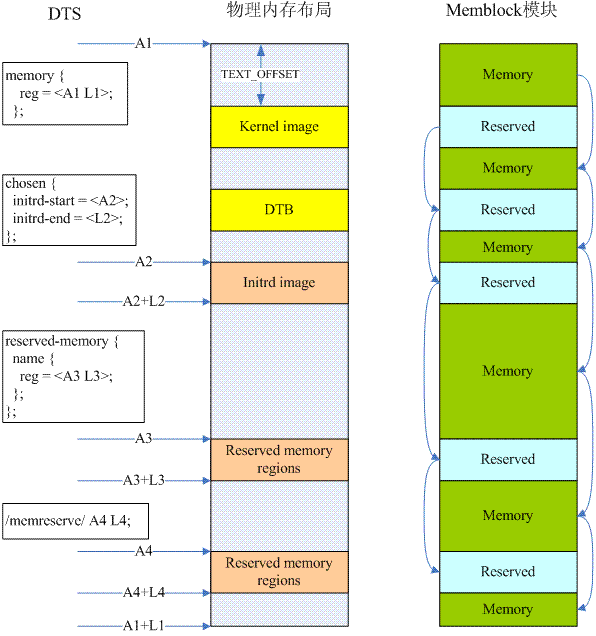
（1）choosen node。该节点有一个bootargs属性，该属性定义了内核的启动参数，而在启动参数中，可能包括了mem=nn[KMG]这样的参数项。initrd-start和initrd-end参数定义了initial ramdisk image的物理地址范围。

（2）memory node。这个节点主要定义了系统中的物理内存布局。主要的布局信息是通过reg属性来定义的，该属性定义了若干的起始地址和size条目。

（3）DTB header中的memreserve域。对于dts而言，这个域是定义在root node之外的一行字符串，例如：/memreserve/ 0x05e00000 0x00100000;，memreserve之后的两个值分别定义了起始地址和size。对于dtb而言，memreserve这个字符串被DTC解析并称为DTB header中的一部分。更具体的信息可以参考[device tree基础](http://www.wowotech.net/device_model/dt_basic_concept.html)文档，了解DTB的结构。

（4）reserved-memory node。这个节点及其子节点定义了系统中保留的内存地址区域。保留内存有两种，一种是静态定义的，用reg属性定义的address和size。另外一种是动态定义的，只是通过size属性定义了保留内存区域的长度，或者通过alignment属性定义对齐属性，动态定义类型的子节点的属性不能精准的定义出保留内存区域的起始地址和长度。在建立地址映射方面，可以通过no-map属性来控制保留内存区域的地址映射关系的建立。

通过对DTB中上述信息的解析，其实内核已经基本对内存布局有数了，但是如何来管理这些信息呢？这也就是著名的memblock模块，主要负责在初始化阶段用来管理物理内存。一个参考性的示意图如下：

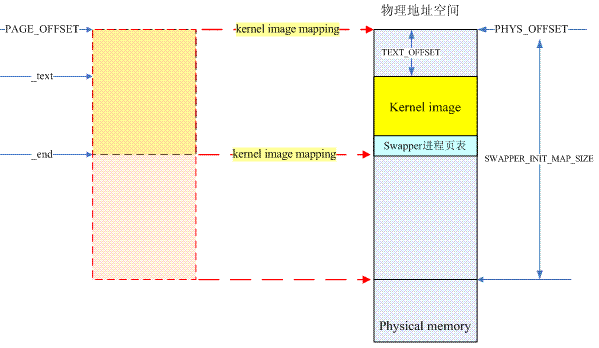


内核在收集了若干和memory相关的信息后，会调用memblock模块的接口API（例如：memblock\_add、memblock\_reserve、memblock\_remove等）来管理这些内存布局的信息。内核需要动态管理起来的内存资源被保存在memblock的memory type的数组中（上图中的绿色block，按照地址的大小顺序排列），而那些需要预留的，不需要内核管理的内存被保存在memblock的reserved type的数组中（上图中的青色block，也是按照地址的大小顺序排列）。

了解到了当前的物理内存的布局，但是内核仍然只是能够访问部分内存（kernel image mapping和DTB那两段内存，上图中黄色block），大部分的内存仍然处于黑暗中，等待光明的到来，也就是说需要创建这些内存的地址映射。

在这个时间点上，创建内存的地址映射有一个悖论：创建地址映射需要分配内存，但是这时候伙伴系统没有ready，无法动态分配。也许你会说，memblock不是已经ready了吗，不可以调用memblock\_alloc进行物理内存的分配吗？当然可以，memblock\_alloc分配的物理内存仍然需要通过虚拟地址访问，而这些内存都还没有创建地址映射，因此内核一旦访问memblock\_alloc分配的物理内存，悲剧就会发生了。

怎么办呢？内核采用了一个巧妙的办法：那就是控制创建地址映射，memblock\_alloc分配页表内存的顺序。也就是说刚开始的时候创建的地址映射不需要页表内存的分配，当内核需要调用memblock\_alloc进行页表物理地址分配的时候，很多已经创建映射的内存已经ready了，这样，在调用create\_mapping的时候不需要分配页表内存。更具体的解释参考下面的图片：



我们知道，在内核编译的时候，在BSS段之后分配了几个page用于swapper进程地址空间（内核空间）的映射，当然，由于kernel image不需要mapping那么多的地址，因此swapper进程translation table的最后一个level中的entry不会全部的填充完毕。换句话说：swapper进程页表可以支持远远大于kernel image mapping那一段的地址区域，实际上，它可以支持的地址段的size是SWAPPER\_INIT\_MAP\_SIZE。为（PAGE\_OFFSET，PAGE\_OFFSET＋SWAPPER\_INIT\_MAP\_SIZE）这段虚拟内存创建地址映射，mapping到（PHYS\_OFFSET，PHYS\_OFFSET＋SWAPPER\_INIT\_MAP\_SIZE）这段物理内存的时候，调用create\_mapping不会发生内存分配，因为所有的页表都已经存在了，不需要动态分配。