2.1 Linux 内核对内存的使用方法

本节首先说明 Linux 0.11 系统中比较直观的物理内存的使用情况，然后分别描述内存的分段和分页管理机制以及 CPU 多任务和保护方式。最后我们再综合说明 Linux 0.11 系统中内核代码和数据以及各个任务的代码和数据在虚拟地址、线性地址和物理地址之间的对应关系。若能充分理解本节的内容，则说明我们对 Intel 80X86 CPU 的内存管理技术已经吃透。

2.1.1 物理内存

在 Linux 0.11 内核中，为了有效地使用机器中的物理内存，在系统初始化阶段内存被划分成几个功能区域，见图 1 所示。

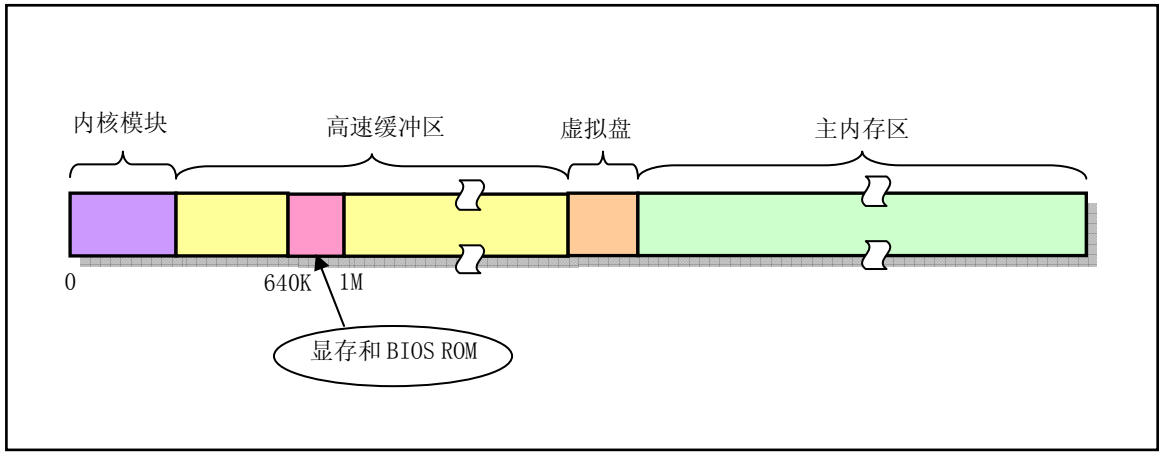

图 1 物理内存使用的功能分布图

其中，Linux 内核程序占据在物理内存的开始部分，接下来是供硬盘或软盘等块设备使用的高速缓冲区部分（其中要扣除显示卡内存和 ROM BIOS 所占用的内存地址范围 640K--1MB）。当一个进程需要读取块设备中的数据时，系统会首先把数据读到高速缓冲区中；当有数据需要写到块设备上去时，系统也是先将数据放到高速缓冲区中，然后由块设备驱动程序写到相应的设备上。内存的最后部分是供所有程序可以随时申请和使用的主内存区。内核程序在使用主内存区时，也同样首先要向内核内存管理模块提出申请，在申请成功后方能使用。对于含有 RAM 虚拟盘的系统，主内存区头部还要划去一部分，供虚拟盘存放数据。

由于计算机系统中所含的实际物理内存容量有限，因此 CPU 中通常都提供了内存管理机制对系统中的内存进行有效的管理。在 Intel 80386 及以后的 CPU 中提供了两种内存管理（变换）系统：内存分段系统（Segmentation System）和分页系统（Paging System）。而分页管理系统是可选择的，由系统程序员通过编程来确定是否采用。为了能有效地使用这些物理内存，Linux 系统同时采用了内存分段和分页管理机制。

2.1.2 内存地址空间概念

Linux 0.11 内核中，在进行地址映射操作时，我们需要首先分清 3 种地址以及它们之间的变换概念：a. 程序（进程）的虚拟和逻辑地址；b. CPU 的线性地址；c. 实际物理内存地址。

虚拟地址（Virtual Address）是指由程序产生的由段选择符和段内偏移地址两个部分组成的地址。因为这两部分组成的地址并没有直接用来访问物理内存，而是需要通过分段地址变换机制处理或映射后才

对应到物理内存地址上，因此这种地址被称为虚拟地址。虚拟地址空间由 GDT 映射的全局地址空间和由 LDT 映射的局部地址空间组成。选择符的索引部分由 13 个比特位表示，加上区分 GDT 和 LDT 的 1 个比特位，因此 Intel 80X86 CPU 共可以索引 16384 个选择符。若每个段的长度都取最大值 4G，则最大虚拟地址空间范围是 16384 * 4G = 64T。

逻辑地址（Logical Address）是指由程序产生的与段相关的偏移地址部分。在 Intel 保护模式下即是指程序执行代码段限长内的偏移地址（假定代码段、数据段完全一样）。应用程序员仅需与逻辑地址打交道，而分段和分页机制对他来说是完全透明的，仅由系统编程人员涉及。

线性地址（Linear Address）是逻辑地址到物理地址变换之间的中间层，是处理器可寻址的内存空间（称为线性地址空间）中的地址。程序代码会产生逻辑地址，或者说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址可以再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。Intel 80386 的线性地址空间容量为 4G。

物理地址（Physical Address）是指出现在 CPU 外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。如果启用了分页机制，那么线性地址会使用页目录和页表中的项变换成物理地址。如果没有启用分页机制，那么线性地址就直接成为物理地址了。

虚拟内存（Virtual Memory）是指计算机呈现出要比实际拥有的内存大得多的内存量。因此它允许程序员编制并运行比实际系统拥有的内存大得多的程序。这使得许多大型项目也能够在具有有限内存资源的系统上实现。一个很恰当的比喻是：你不需要很长的轨道就可以让一列火车从上海开到北京。你只需要足够长的铁轨（比如说 3 公里）就可以完成这个任务。采取的方法是把后面的铁轨立刻铺到火车的前面，只要你的操作足够快并能满足要求，列车就能象在一条完整的轨道上运行。这也就是虚拟内存管理需要完成的任务。在 Linux 0.11 内核中，给每个程序（进程）都划分了总容量为 64MB 的虚拟内存空间。因此程序的逻辑地址范围是 0x0000000 到 0x4000000。

有时我们也把逻辑地址称为虚拟地址。因为与虚拟内存空间的概念类似，逻辑地址也是与实际物理内存容量无关的。

2.1.3 内存分段机制

在内存分段系统中，一个程序的逻辑地址是通过分段机制自动地映射（变换）到中间层的 4GB（2^{32}）线性地址空间中。每次对内存的引用都是对内存段中内存的引用。当程序引用一个内存地址时，通过把相应的段基址加到程序员看得见的逻辑地址上就形成了一个对应的线性地址。此时若没有启用分页机制，则该线性地址就被送到 CPU 的外部地址总线上，用于直接寻址对应的物理内存。见图 2 所示。

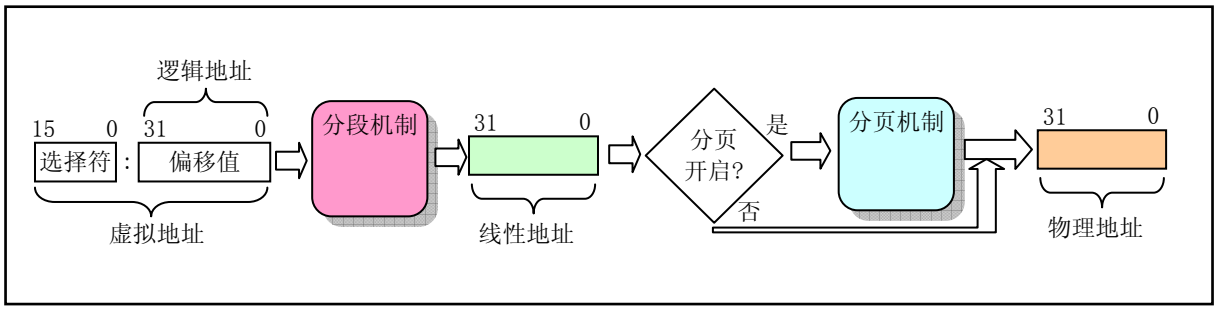

图 2 虚拟地址（逻辑地址）到物理地址的变换过程

若采用了分页机制，则此时线性地址只是一个中间结果，还需要使用分页机制进行变换，再最终映射到实际物理内存地址上。与分段机制类似，分页机制允许我们重新定向（变换）每次内存引用，以适应我们的特殊要求。使用分页机制最普遍的场合是当系统内存实际上被分成很多凌乱的块时,它可以建立一个大而连续的内存空间的映像，好让程序不用操心和管理这些分散的内存块。分页机制增强了分段机制的性能。页地址变换是建立在段变换基础之上的。任何分页机制的保护措施并不会取代段变换的保护措施而只

是进行更进一步的检查操作。

因此，CPU 进行地址变换（映射）的主要目的是为了解决虚拟内存空间到物理内存空间的映射问题。虚拟内存空间的含义是指一种利用二级或外部存储空间，使程序能不受实际物理内存量限制而使用内存的一种方法。通常虚拟内存空间要比实际物理内存量大得多。

那么虚拟内存空间管理是怎样实现的呢？原理与上述列车运行的比喻类似。首先，当一个程序需要使用一块不存在的内存时（也即在内存页表项中已标出相应内存页面不在内存中），CPU 就需要一种方法来得知这个情况。这是通过 80386 的页错误异常中断来实现的。当一个进程引用一个不存在页面中的内存地址时，就会触发 CPU 产生页出错异常中断，并把引起中断的线性地址放到 CR2 控制寄存器中。因此处理该中断的过程就可以知道发生页异常的确切地址，从而可以把进程要求的页面从二级存储空间（比如硬盘上）加载到物理内存中。如果此时物理内存已经被全部占用，那么可以借助二级存储空间的一部分作为交换缓冲区（Swapper）把内存中暂时不使用的页面交换到二级缓冲区中，然后把要求的页面调入内存中。这也就是内存管理的缺页加载机制，在 Linux 0.11 内核中是在程序 mm/memory.c 中实现。

Intel CPU 使用段（Segment）的概念来对程序进行寻址。每个段定义了内存中的某个区域以及访问的优先级等信息。现在我们假定大家知晓实模式下内存寻址原理的基础上，根据 CPU 在实模式和保护模式下寻址方式的不同，用比较的方法来简单说明 32 位保护模式运行机制下内存寻址的主要特点。在后续章节中将逐步对其进行详细说明。

在实模式下，寻址一个内存地址主要是使用段和偏移值，段值被存放在段寄存器中（例如 ds），并且段的长度被固定为 64KB。段内偏移地址存放在任意一个可用于寻址的寄存器中（例如 si）。因此，根据段寄存器和偏移寄存器中的值，就可以算出实际指向的内存地址，见图 3 (a)所示。

而在保护模式运行方式下，段寄存器中存放的不再是被寻址段的基地址，而是一个段描述符表（Segment Descriptor Table）中某一描述符项在表中的索引值。索引值指定的段描述符项中含有需要寻址的内存段的基地址、段的长度值和段的访问特权级别等信息。寻址的内存位置是由该段描述符项中指定的段基地址值与一个段内偏移值组合而成。段的长度可变，由描述符中的内容指定。可见，和实模式下的寻址相比，段寄存器值换成了段描述符表中相应段描述符的索引值以及段表选择位和特权级，称为段选择符（Segment Selector），但偏移值还是使用了原实模式下的概念。这样，在保护模式下寻址一个内存地址就需要比实模式下多一道手续，也即需要使用段描述符表。这是由于在保护模式下访问一个内存段需要的信息比较多，而一个 16 位的段寄存器放不下这么多内容。示意图见图 3 (b)所示。注意，如果你不在一个段描述符中定义一个内存线性地址空间区域，那么该地址区域就完全不能被寻址，CPU 将拒绝访问该地址区域。

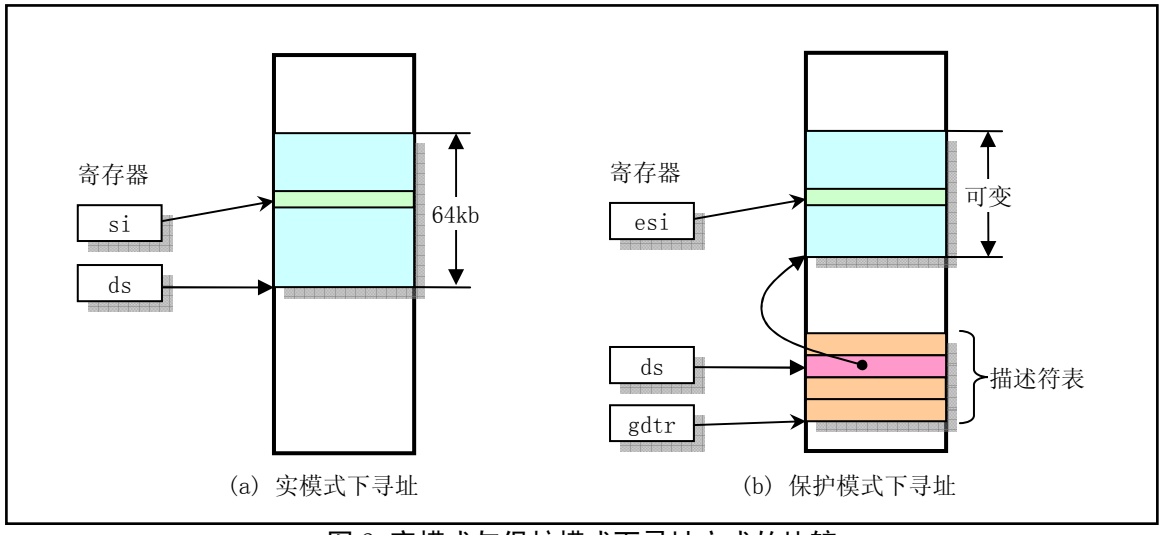

图 3 实模式与保护模式下寻址方式的比较

段描述符表中一个描述符项的具体格式见图 4 所示。每个描述符占用 8 个字节，其中含有所描述段在线性地址空间中的起始地址（基址）、段的长度、段的类型（例如代码段和数据段）、段的特权级别和其他一些信息。一个段可以定义的最大长度是 4GB。各个字段的具体含义将在后续章节加以说明。

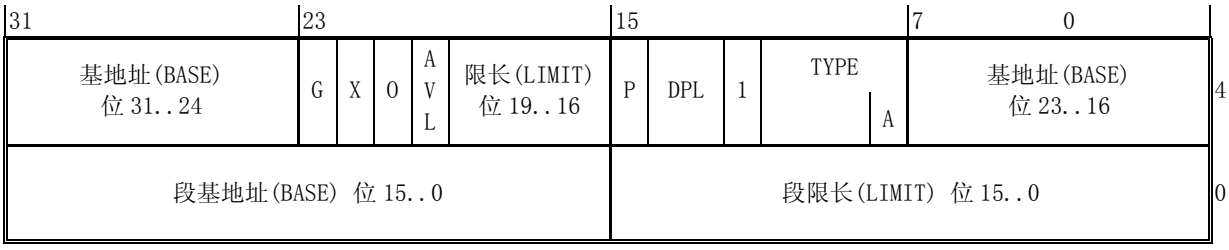

图 4 程序代码段和数据段的描述符格式

保存描述符项的描述符表有 3 种类型，每种用于不同目的。全局描述符表 GDT（Global Descriptor Table)是主要的基本描述符表,该表可被所有程序用于引用访问一个内存段。中断描述符表 IDT(Interrupt Descriptor Table）保存有定义中断或异常处理过程的段描述符。IDT 表直接替代了 8086 系统中的中断向量表。为了能在 80X86 保护模式下正常运行，我们必须为 CPU 定义一个 GDT 表和一个 IDT 表。最后一种类型的表是局部描述符表 LDT（Local Descriptor Table)。该表应用于多任务系统中，通常每个任务使用一个 LDT 表。作为对 GDT 表的扩充，每个 LDT 表为对应任务提供了更多的可用描述符项，因而也为每个任务提供了可寻址内存空间的范围。这些表可以保存在线性地址空间的任何地方。为了让 CPU 能定位 GDT 表、IDT 表和当前的 LDT 表，需要为 CPU 分别设置 GDTR、IDTR 和 LDTR 三个特殊寄存器。这些寄存器中将存储对应表的 32 位线性基地址和表的限长字节值。表限长值是表的长度值-1。

但 CPU 要寻址一个段时，就会使用 16 位的段寄存器中的选择符（有些书上称为选择子）来定位一个段描述符，选择符的格式见图 5 所示。在 80X86 CPU 中，段寄存器中的值右移 3 位即是描述符表中一个描述符的索引值。13 位的索引值最多可定位 8192（0--8191）个的描述符项。选择符中位 2（TI）用来指定使用哪个表。若该位是 0 则选择符指定的是 GDT 表中的描述符，否则是 LDT 表中的描述符。

图 5 段选择符格式

每个程序都可有若干个内存段组成。程序的逻辑地址（或称为虚拟地址）即是用于寻址这些段和段中具体地址位置。在 Linux 0.11 中，程序逻辑地址到线性地址的变换过程使用了 CPU 的全局段描述符表 GDT 和局部段描述符表 LDT。由 GDT 映射的地址空间称为全局地址空间，由 LDT 映射的地址空间则称为局部地址空间，而这两者构成了虚拟地址的空间。具体的使用方式见图 6 所示。

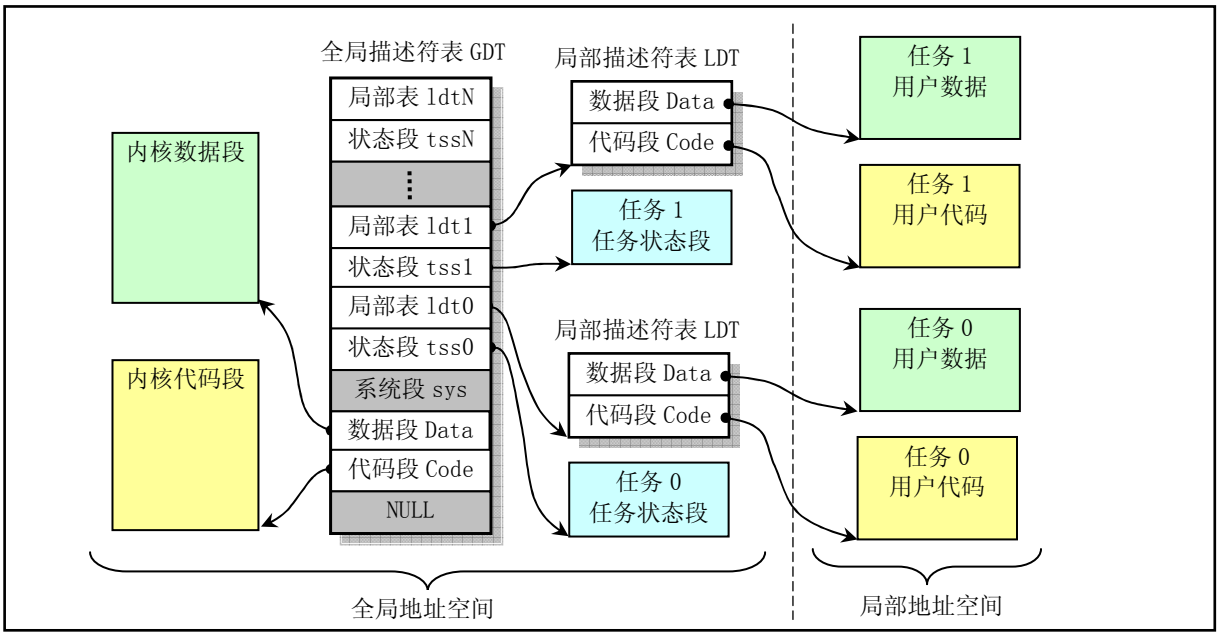

图 6 Linux 系统中虚拟地址空间分配图

图中画出了具有两个任务时的情况。对于中断描述符表 idt，它是保存在内核代码段中的。由于在 Linux 0.11 内核中，内核和各任务的代码段和数据段都分别被映射到线性地址空间中相同基址处，且段限长也一样，因此内核和任务的代码段和数据段都分别是重叠的。另外，Linux 0.11 内核中没有使用系统段描述符。

2.1.4 内存分页管理

内存分页管理机制的基本原理是将 CPU 整个线性内存区域划分成 4096 字节为 1 页的内存页面。程序申请使用内存时，就以内存页为单位进行分配。内存分页机制的实现方式与分段机制很相似，但并不如分段机制那么完善。因为分页机制是在分段机制之上实现的，所以其结果是对系统内存具有非常灵活的控制权，并且在分段机制的内存保护上更增加了分页保护机制。为了在 80X86 保护模式下使用分页机制，需要把控制寄存器 CR0 的最高比特位（位 31）置位。

在使用这种内存分页管理方法时，每个执行中的进程（任务）可以使用比实际内存容量大得多的连续地址空间。为了在使用分页机制的条件下把线性地址映射到容量相对很小的物理内存空间上，80386 使用了页目录表和页表。页目录表项与页表项格式基本相同，都占用 4 个字节，并且每个页目录表或页表必须只能包含 1024 个页表项。因此一个页目录表或一个页表分别共占用 1 页内存。页目录项或页表项的格式见图 7 所示。其中各字段的详细说明请参见第 10 章（内存管理）内容。页目录项和页表项的小区别在于页表项有个已写位 D（Dirty），而页目录项则没有。

31 12	11 9	8	7	6	5	4 3	2	1	0
页框地址 位 31..12 (PAGE FRAME ADDRESS)	可用 (AVAIL)	0	D	A	0	0 0	U / S	R / W	P

图 7 页目录和页表表项结构

线性地址到物理地址的变换过程见图 8 所示。图中控制寄存器 CR3 保存着是当前页目录表在物理内存中的基地址。32 位的线性地址被分成三个部分，分别用来在页目录表和页表中定位对应的页目录项和页表项以及在对应的物理内存页面中指定页面内的偏移位置。

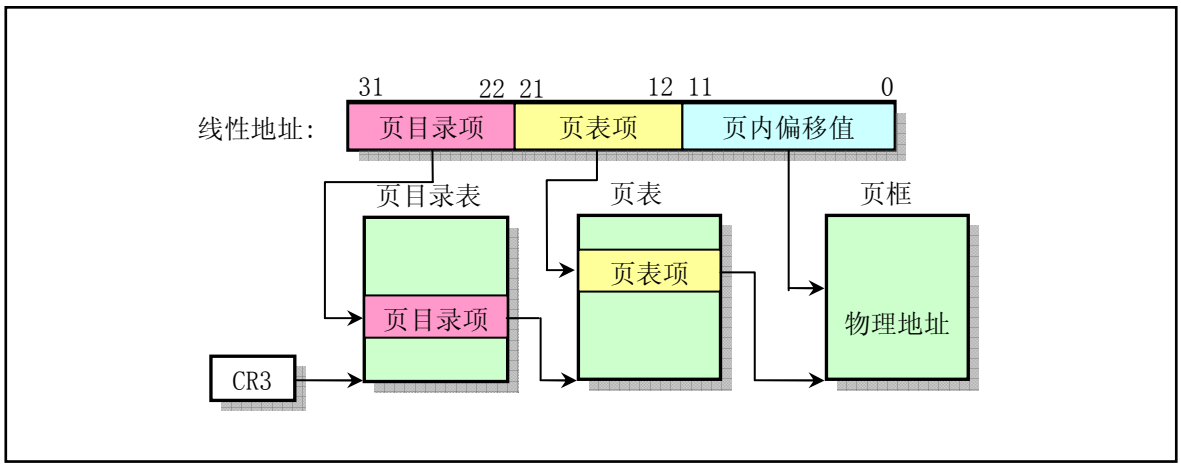

图 8 线性地址到物理地址的变换示意图

对于 Intel 80386 系统，其 CPU 可以提供多达 4G 的线性地址空间。一个任务的虚拟地址需要首先通过其局部段描述符变换为 CPU 整个线性地址空间中的地址，然后再使用页目录表 PDT（一级页表）和页表 PT（二级页表）映射到实际物理地址页上。因此两种变换不能混淆。

为了使用实际物理内存，每个进程的线性地址通过二级内存页表动态地映射到主内存区域的不同内存页上。因此每个进程最大可用的虚拟内存空间是 64MB。每个进程的逻辑地址通过加上(任务号-1)*64MB，即可转换为线性空间中的地址。不过在注释中，在不至于搞混的情况下我们通常将进程中的此类地址简单地称为逻辑地址或线性地址。有关内存分页管理的详细信息，请参见第 10 章开始部分的有关说明，或参见附录。

对于 Linux 0.11 内核，系统设置全局描述符表 GDT 中的段描述符项数最大为 256，其中 2 项空闲、2 项系统使用，每个进程使用两项。因此，此时系统可以最多容纳(256-4)/2 + 1=127 个任务，并且虚拟地址范围是 ((256-4)/2)* 64MB 约等于 8G。但 0.11 内核中人工定义最大任务数 NR_TASKS = 64 个，每个任务逻辑地址范围是 64M，并且各个任务在线性地址空间中的起始位置是(任务号-1)*64MB。因此全部任务所使用的线性地址空间范围是 64MB*64 =4G，见图 9 所示。图中示出了当系统具有 4 个任务时的情况。内核代码段和数据段被映射到线性地址空间的开始 16MB 部分，并且代码和数据段都映射到同一个区域，完全相重叠。而第 1 个任务（任务 0）是由内核“人工”启动运行的，其代码和数据包含在内核代码和数据中，因此该任务所占用的线性地址空间范围比较特殊。任务 0 的代码段和数据段的长度是从线性地址 0 开始的 640KB 范围，其代码和数据段也是完全重叠的，并且与内核代码段和数据段有重叠的部分。因此实际上任务 0 的线性地址范围与图中所示有差异。任务 1 的线性地址空间范围也只有从 64MB 开始的 640KB 长度。它们之间的详细对应关系见后面说明。任务 2 和任务 3 分别被映射线性地址 128MB 和 192MB 开始的地方，并且它们的逻辑地址范围均是 64MB。由于 4G 地址空间范围正好是 CPU 的线性地址空间范围和可寻址的最大物理地址空间范围，而且在把任务 0 和任务 1 的逻辑地址范围看作 64MB 时，系统中同时可有任务的逻辑地址范围总和也是 4GB，因此在 0.11 内核中比较容易混淆三种地址概念。

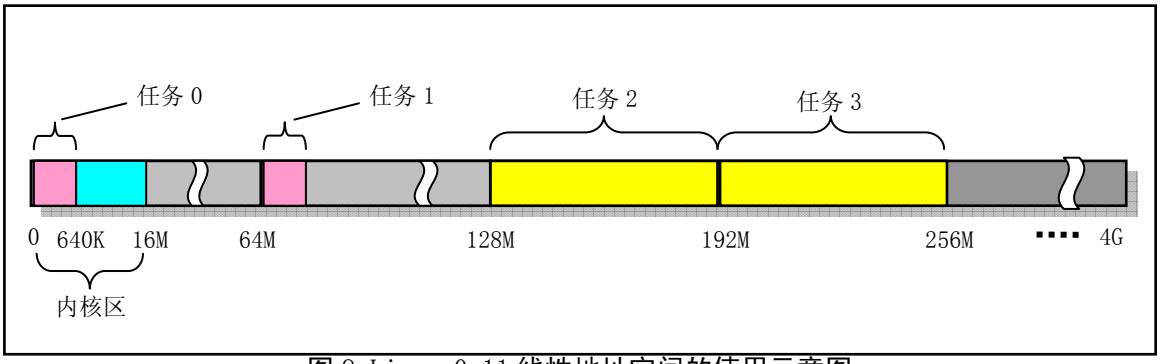

图 9 Linux 0.11 线性地址空间的使用示意图

如果也按照线性空间中任务的排列顺序排列虚拟空间中的任务，那么我们可以有图 10 所示的系统同时可拥有的所有任务在虚拟地址空间中的示意图。其中没有考虑内核代码和数据在虚拟空间中所占用的范围。另外，在图中对于进程 2 和进程 3 还分别给出了各自逻辑空间中代码段和数据段（包括数据和堆栈内容）的位置示意图。

注意，进程逻辑地址空间中的代码段和数据段的概念与 CPU 分段机制中的代码段和数据段不是同一个概念。CPU 分段机制中段的概念确定了在线性地址空间中一个段的用途以及被执行或访问的约束和限制，每个段可以设置在 4GB 线性地址空间中的任何地方，它们可以相互独立也可以完全重叠或部分重叠。而进程在逻辑地址空间中的代码段和数据段则是指由编译器在编译程序和操作系统在加载程序时规定的在进程逻辑空间中顺序排列的代码区域、初始化和未初始化的数据区域以及堆栈区域。

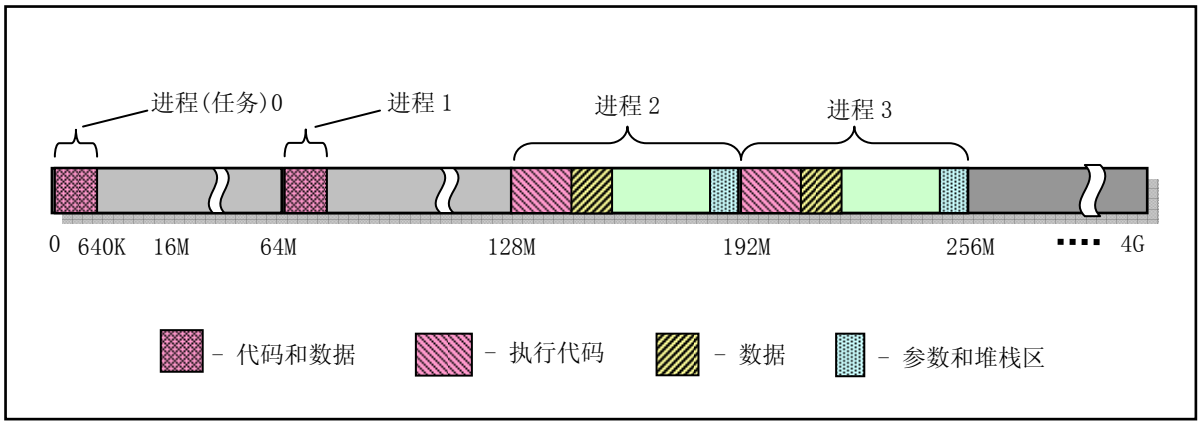

图 10 Linux 0.11 系统任务在虚拟空间中顺序排列所占的空间范围

2.1.5 CPU 多任务和保护方式

Intel 80X86 CPU 共分 4 个保护级，0 级具有最高优先级，而 3 级优先级最低。Linux 0.11 操作系统使用了 CPU 的 0 和 3 两个保护级。内核代码本身会由系统中的所有任务共享。而每个任务则都有自己的代码和数据区，这两个区域保存于局部地址空间，因此系统中的其他任务是看不见的（不能访问的）。而内核代码和数据是由所有任务共享的，因此它保存在全局地址空间中。图 11 给出了这种结构的示意图。图中同心圆代表 CPU 的保护级别（保护层），这里仅使用了 CPU 的 0 级和 3 级。而径向射线则用来区分系统中的各个任务。每条径向射线指出了各任务的边界。除了每个任务虚拟地址空间的全局地址区域，任务 1 中的地址与任务 2 中相同地址处是无关的。

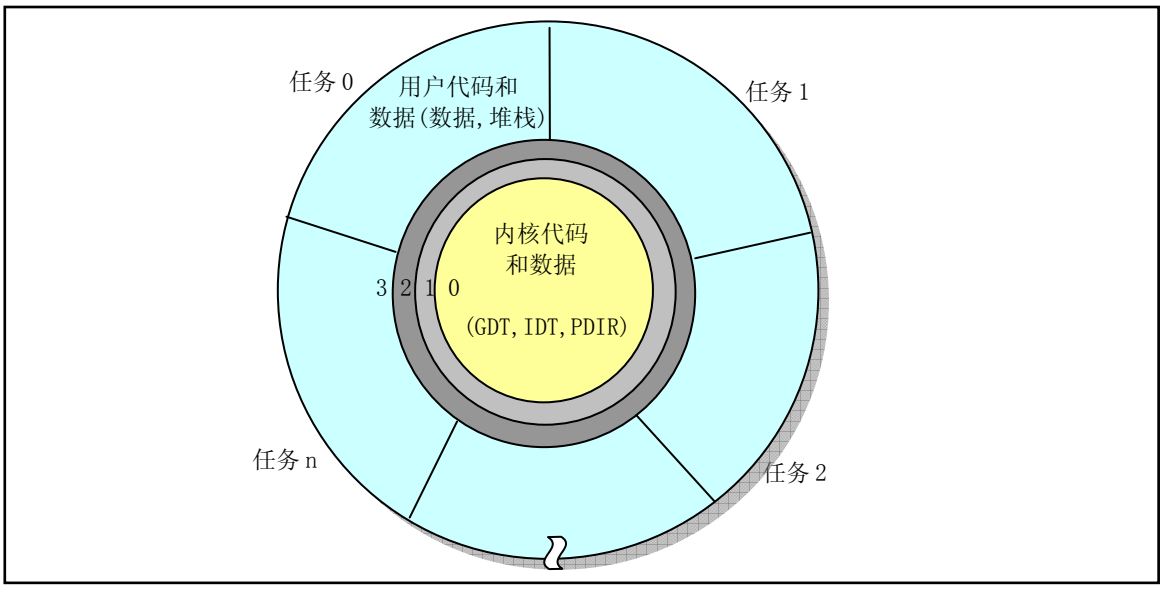

图 11 多任务系统

2.1.6 虚拟地址、线性地址和物理地址之间的关系

前面我们根据内存分段和分页机制详细说明了 CPU 的内存管理方式。现在我们以 Linux 0.11 系统为例，详细说明内核代码和数据以及各任务的代码和数据在虚拟地址空间、线性地址空间和物理地址空间中的对应关系。由于任务 0 和任务 1 的生成或创建过程比较特殊，我们将对它们分别进行描述。

2.1.6.1 内核代码和数据的地址

对于 Linux 0.11 内核代码和数据来说，在 head.s 程序的初始化操作中已经把内核代码段和数据段都设置成为长度为 16MB 的段。在线性地址空间中这两个段的范围重叠，都是从线性地址 0 开始到地址 0xFFFFFF 共 16MB 地址范围。在该范围中含有内核所有的代码、内核段表（GDT、IDT、TSS）、页目录表和内核的二级页表、内核局部数据以及内核临时堆栈（将被用作第 1 个任务（任务 0）的用户堆栈）。其页目录表和二级页表已设置成把 0--16MB 的线性地址空间一一对应到物理地址上，占用了 4 个目录项，即 4 个二级页表。因此对于内核代码或数据的地址来说，我们可以直接把它们看作是物理内存中的地址。此时内核的虚拟地址空间、线性地址空间和物理地址空间三者之间的关系可用图 12 来表示。

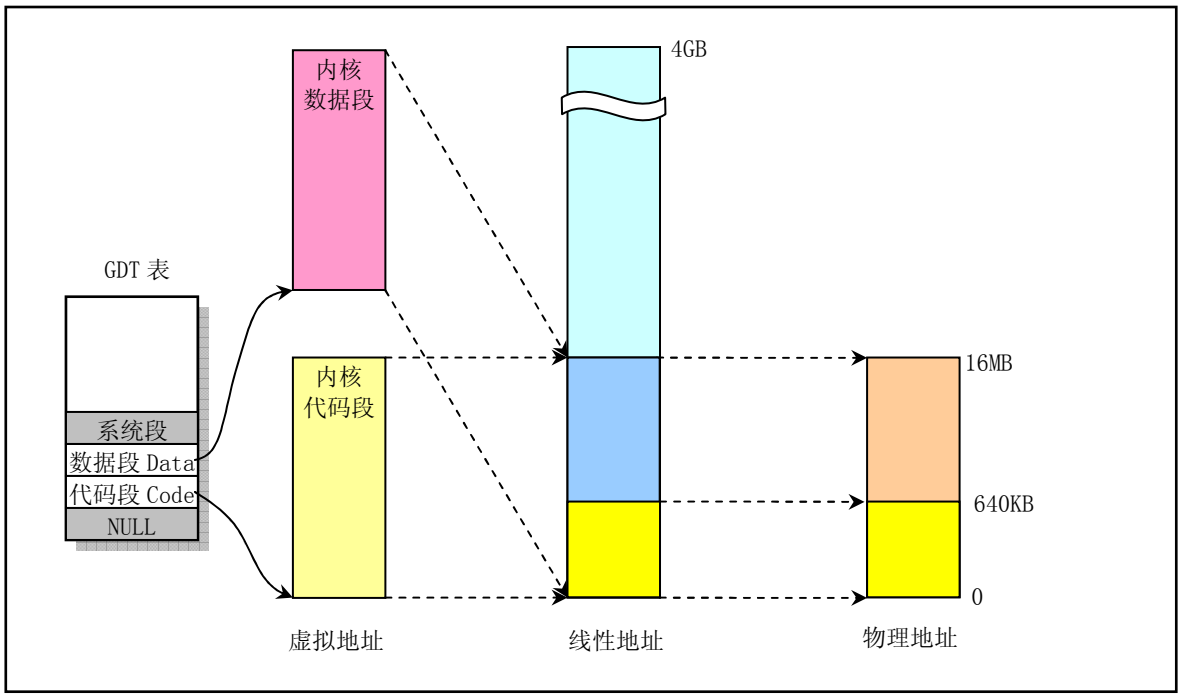

图 12 内核代码和数据段在三种地址空间中的关系

默认情况下，Linux 0.11 内核可管理 16MB 的物理内存，共有 4096 个物理页面（页帧），每个页面 4KB。通过上述分析可以看出：①内核代码段和数据段区域在线性地址空间和物理地址空间中是一样的。这样设置可以大大简化内核的初始化操作。②GDT 和 IDT 在内核数据段中，因此它们的线性地址也同样等于它们的物理地址。否则的话，在进入保护模式和开启分页机制时这两个表就需要被重新建立或移动位置，描述符也需要重新加载。③除任务 0 以外的其他任务所需要的物理内存页面与线性地址中的不同，因此内核需要动态地在主内存区中为它们作映射操作，动态地建立页目录项和页表项。

2.1.6.2 任务 0 的地址对应关系

任务 0 是系统中一个人工启动的第一个任务。它的代码段和数据段长度被设置为 640KB。该任务的代码和数据直接包含在内核代码和数据中，是从线性地址 0 开始的 640KB 内容，因此可以它直接使用内核代码已经设置好的页目录和页表进行分页地址变换。同样它的代码和数据段在线性地址空间中也是重叠的。对应的任务状态段 TSS0 也是手工预设置好的，并且位于任务 0 数据结构信息中，参见 sched.h 第 113 行开始的数据。TSS0 段位于内核 sched.c 程序的代码中，长度为 104 字节。三个地址空间中的映射对应关系见图 13 所示。

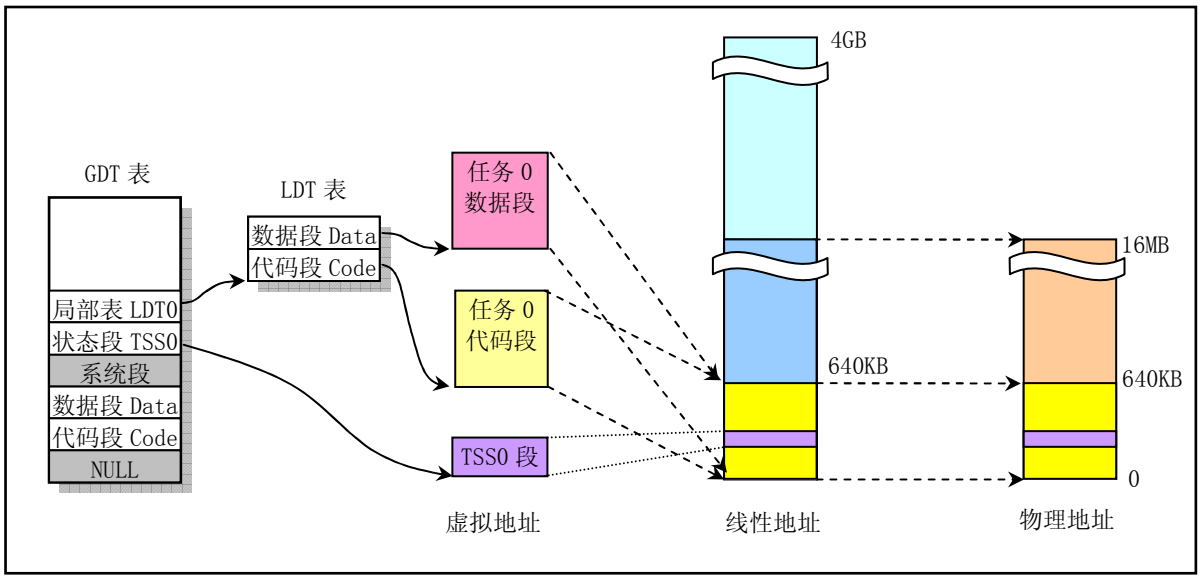

图 13 任务 0 在三个地址空间中的相互关系

由于任务 0 直接被包含在内核代码中，因此不需要为其再另外分配内存页。它运行时所需要的内核态堆栈和用户态堆栈空间都在内核代码区中，用户态堆栈将与任务 1 共享使用，但用户态堆栈将全部留给任务 1 使用。见下节中的说明。

2.1.6.3 任务 1 的地址对应关系

与任务 0 类似，任务 1 也是一个特殊的任务。它的代码也在内核代码区域中。与任务 0 不同的是在线性地址空间中，系统在使用 fork()创建任务 1（init 进程）时为任务 1 的二级页表在主内存区申请了一页内存来存放，并复制了父进程（任务 0）的页目录和二级页表项。因此任务 1 的长度也是 640KB，并且其代码段和数据段相重叠，只占用一个页目录项和一个二级页表。另外，在 fork()创建任务 1 时，系统还会为任务 1 在主内存申请一页内存用来存放它的任务数据结构和用作任务 1 的内核堆栈空间。任务数据结构信息中包括任务 1 的 TSS 段结构信息。见图 14 所示。

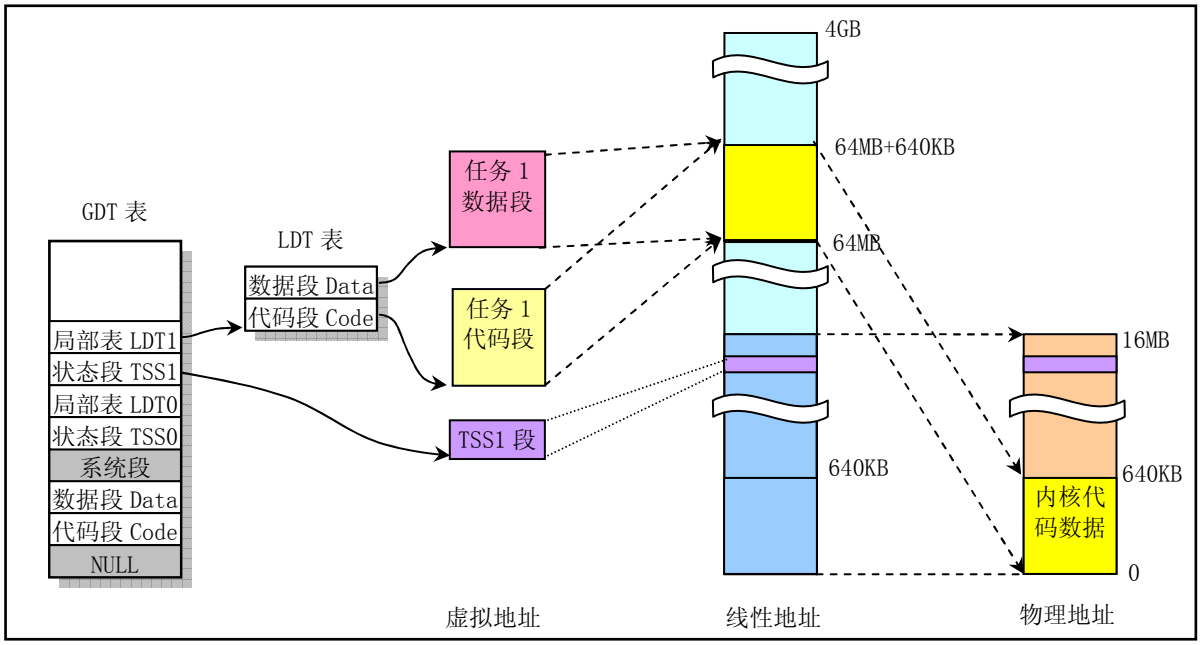

图 14 任务 1 在三种地址空间中的关系

任务 1 的用户态堆栈空间将直接使用处于内核代码和数据区域（线性地址 0--640KB）中任务 0 的用户态堆栈空间。

2.1.6.4 其他任务的地址对应关系

这里，其他任务指的是被创建的从任务 2 开始的任务。它们的父进程都是 init 进程。在 Linux 0.11 系统中共可以有 64 个进程同时存在。下面以任务 2 为例来说明其他任何任务对地址空间的使用情况。

从任务 2 开始，如果任务号以 nr 来表示，那么任务 nr 在线性地址空间中的起始位置将被设定在 nr*64MB 处。例如任务 2 的开始位置= nr*64MB = 2 * 64MB = 128MB。任务代码段和数据段的最大长度被设置为 64MB，因此任务 2 占有的线性地址空间范围是 128MB--192MB，共占用 64MB/4MB = 16 个页目录项。虚拟空间中任务代码段和数据段都被映射到线性地址空间相同的范围，因此它们也完全重叠。图 15 显示出了任务 2 的代码段和数据段在三种地址空间中的对应关系。

在 Linux 0.11 系统中，任务 2 在被创建出来之后，将运行 execve()函数来执行 shell 程序。图 15 给出的是任务 2 原先复制任务 1 的代码和数据被 shell 的代码段和数据段替换后的情况。这里请注意的是，在执行 execve()函数时，系统虽然在线性地址空间为任务 2 分配了 64MB 的空间，但是并不会立刻为其分配物理内存页面。只有当任务 2 开始执行时由于发生缺页而引起异常时才会由内存管理程序为其在主内存区中分配并映射一页物理内存到线性地址空间中。图中显示出映射了一页物理内存页的情况。

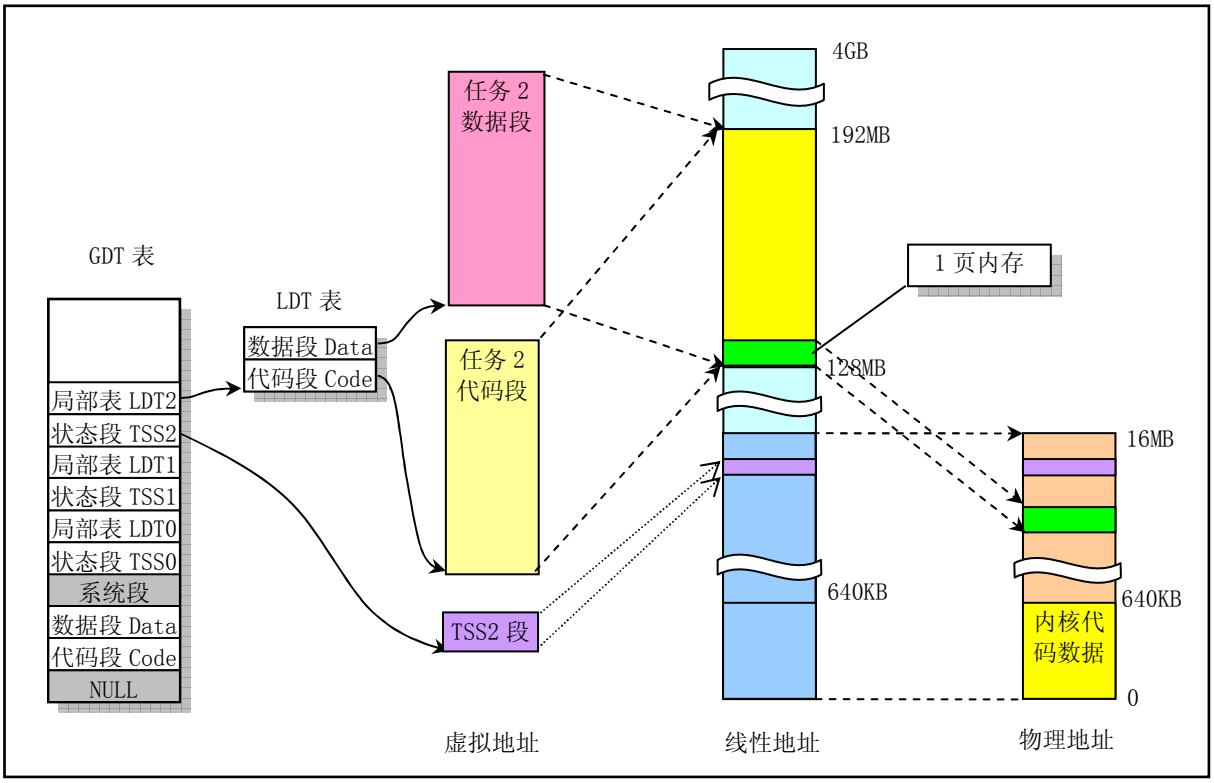

图 15 其他任务地址空间中的对应关系

从 Linux 内核 0.99 版以后，对内存空间的使用方式发生了变化。每个进程可以单独享用整个 4G 的地址空间范围。如果我们能理解本节说描述的内存管理概念，那么对于现在所使用的 Linux 2.x 内核中所使用的内存管理原理也能立刻明白。由于篇幅所限，这里对此不再说明。