**虚拟化，起洞！！！！！**

开始进入虚拟内存

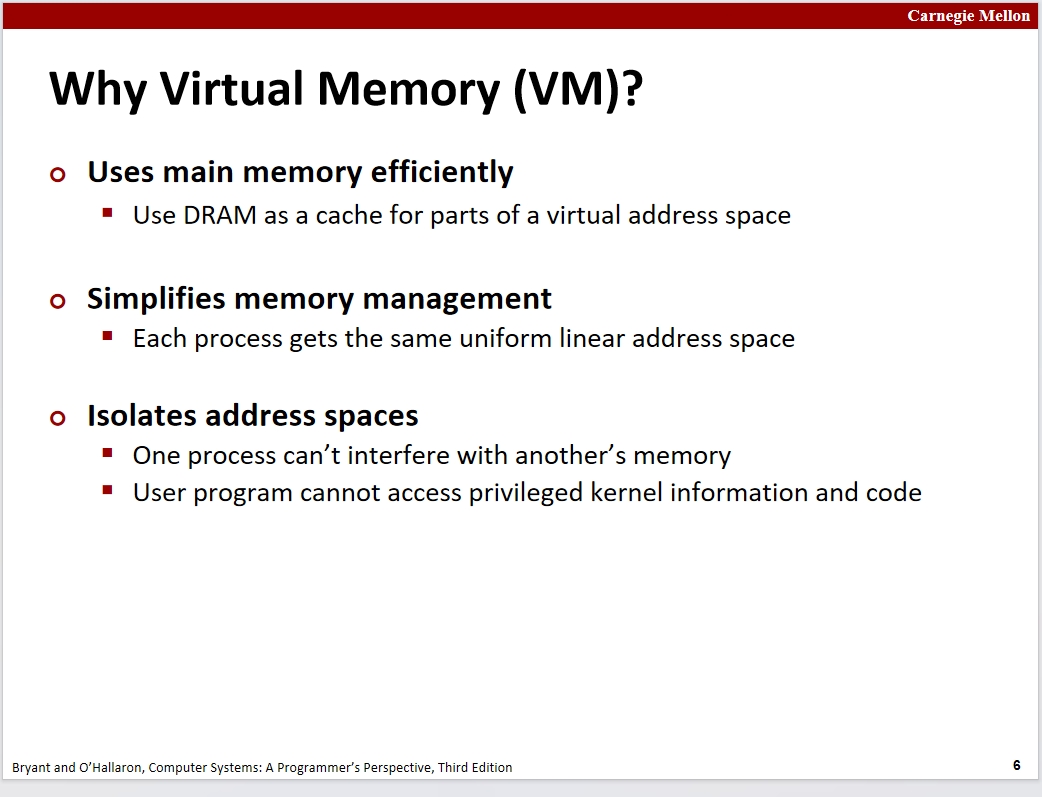
先介绍下几个概念

1~Linear address space。**地址空间**，这是一个地址的集合，这个不是数据字节的集合，而是一个系统的地址的集合。线性地址空间是一个连续的非负整数集合

2~Virtual address space。**虚拟地址空间**，虚拟地址空间是一个包含N=2^n个虚拟地址的结合，是一个线性的地址空间。

3~Physical address space。**物理地址空间**，物理地址空间是一个M=2^m个物理地址的集合。

通常来说虚拟地址空间会比物理空间大的多，即N>>M。物理地址空间对应着系统中实际拥有的DRAM容量。



对于虚拟内存的意义：

1~虚拟内存使用DRAM作为储存在磁盘上的实际数据的缓存。这样能够提高系统对数据的处理效率

2~虚拟内存能够简化我们使用内存的步骤。对于一个程序来说，其一般是不必要使用到很大的一个内存空间的，虚拟地址能够实现对程序使用的内存空间的分配，将一个进程的操作范围限定，使得操作只在一定的范围内，仅使用一部分的虚拟地址空间就能够有效的使用内存。

就比如，我们通常在一个程序运行时值使用到一小块的DRAM。这种架构使得系统能够更好的理由DRAM的容量。

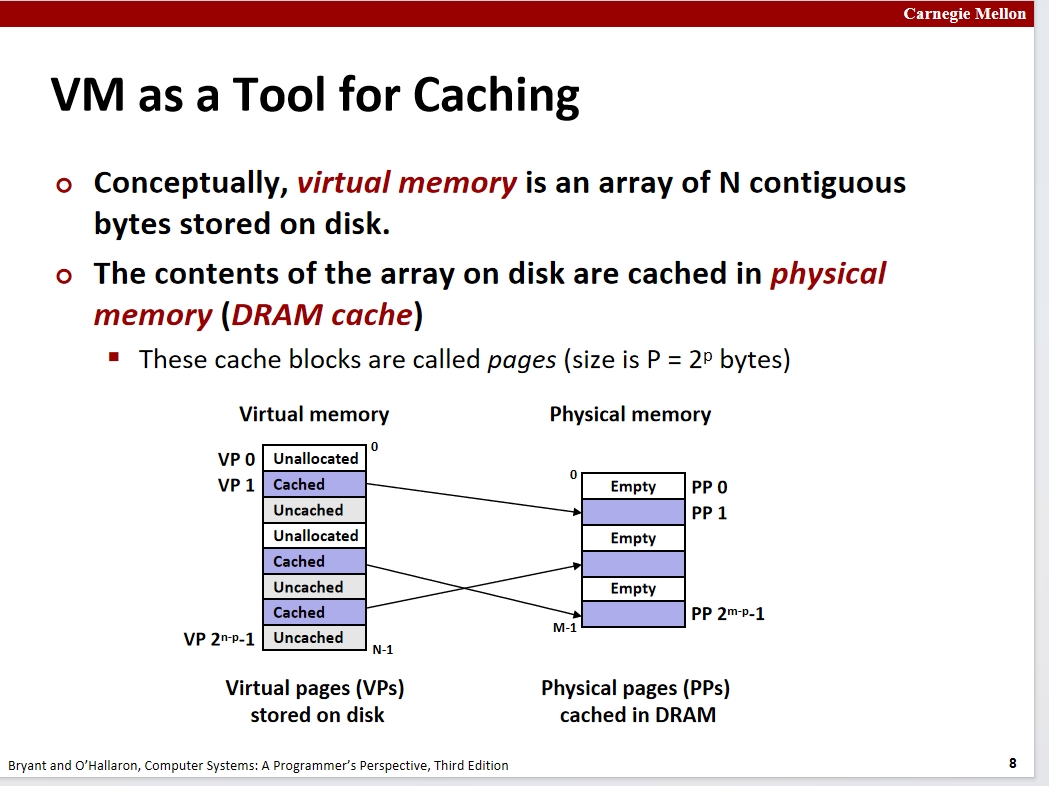
3~虚拟内存能够极大的简化系统对于内存的管理，就比如，我们执行一个进程时，每个进程都具有一个相同外观的内存空间，这些空间内的内存架构通常都是一致的，只是各块的大小不同而已，这种相似的架构使得只需要设计好一种通用的内存管理模式就能够对所有的架构进行管理了。

实际上，对于这些进程对应的虚拟内存块中的数据，可能是分散在主存上的各个部分的，这中虚拟内存的架构也能够更好的利用主存。

3~这种虚拟地址的架构允许了对于访存的保护。在我们看来，系统为一个进程提供了一个独立的地址空间，这个空间不允许其他进程等的访问和修改。这是虚拟内存提供的一种保护机制。

**地址转换，起洞**

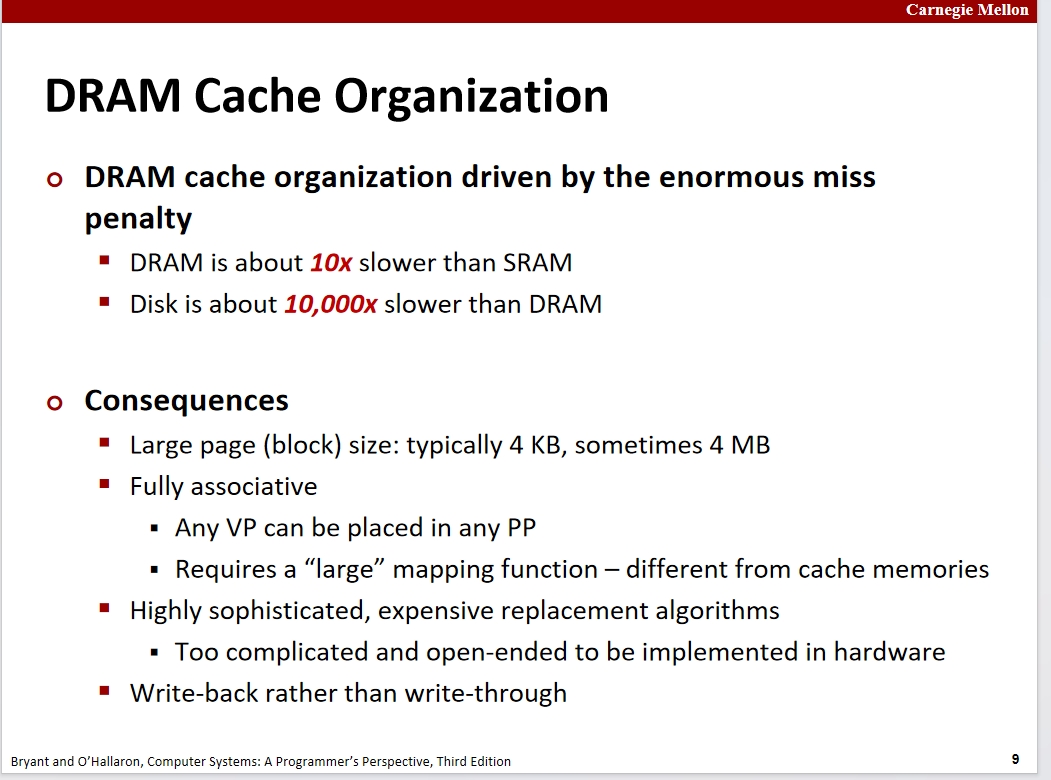
首先，如前面所说，储存在外存(磁盘等)上的虚拟地址内容会缓存在DRAM中。DRAM可视为储存在磁盘上的连续字节数组的缓存。正如前面提到的可执行文件，当文件要执行时，会被加载到主存中执行。既然是缓存，这个主存也有像之前的L1级缓存等一样的分块机制。不过这里的块比之前的缓存块要大的多，毕竟它的作用是要来储存运行文件的。



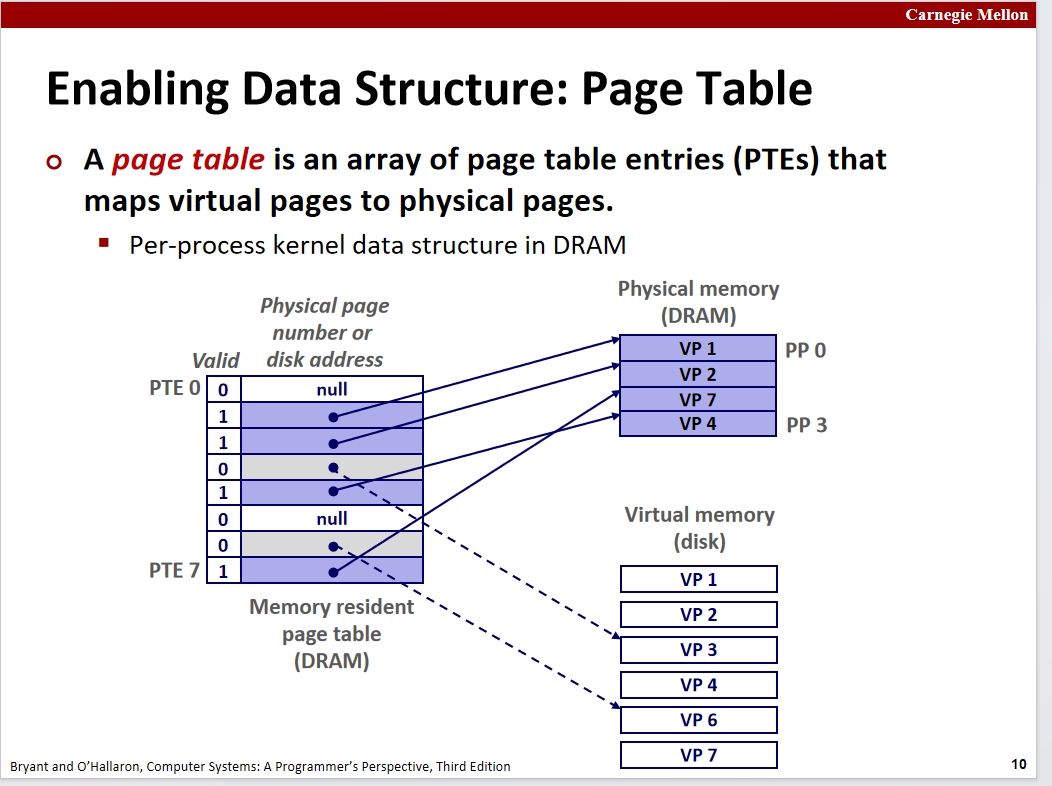
对于虚拟内存，我们可以将其看为储存在外存中的一系列的页面，这些页面每一个都会标识一个数字，由0开始编号。对于这些页面，有些将会被缓存到物理的DRAM缓存器中。除此之外，还有一些映射来告诉我们这些虚拟内存页中有哪些是已经被缓存了的。但是一般来说，虚拟内存上已经被缓存的编号与主存上的块编号间没有必然的联系。

对于哪些没有被缓存的内存块，这些还是被保存到外存的内存块中。除此之外，还有大块大块的未被使用的内存块，毕竟你也不想看你的盘总是爆红吧。

接下来需要分析下这种作为缓存的DRAM为什么会与之前的缓存结构存在重大差别，这是由于这种缓存的特殊性决定的



将DRAM视为缓存，当DRAM未命中时，这就需要从内存中加载数据，这通常需要很多的时间。对于缓存的大小，需要慎重考虑，如果太大，可能就会导致一个块中的数据很多时候不会被使用，浪费了宝贵的主存空间。如果太小，可能就需要频繁地从外存中加载，浪费很多的时间。

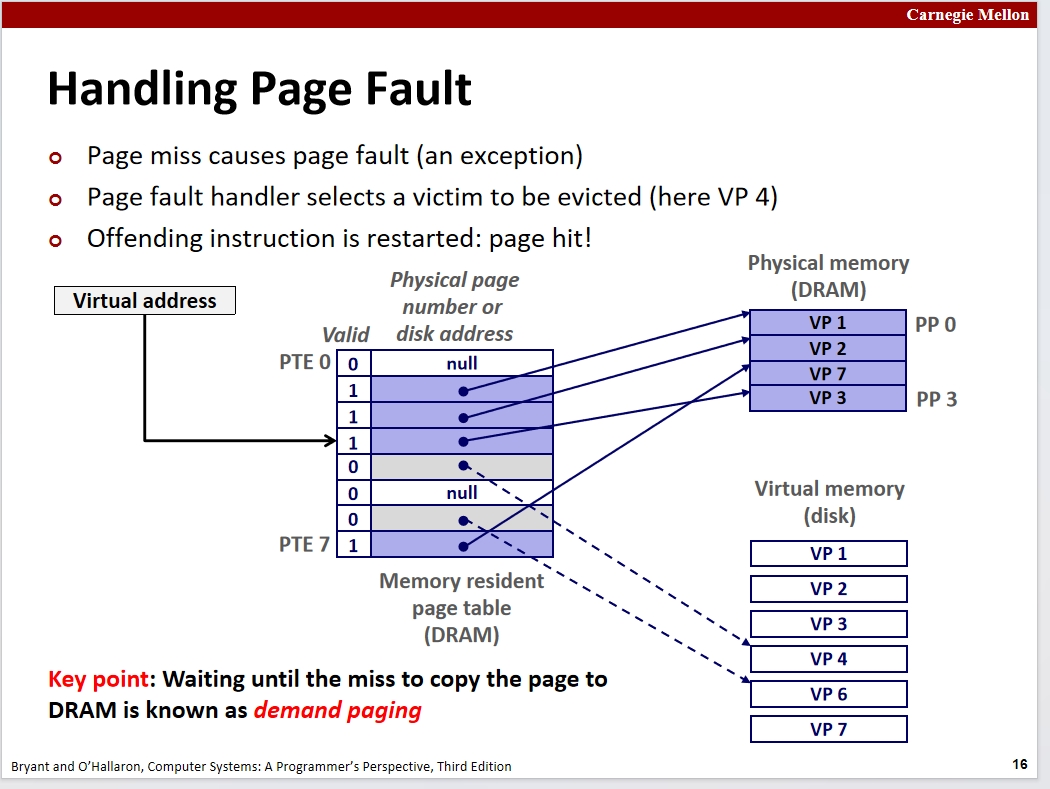


接下来来看记录虚拟页面位置的数据结构（**页表**）这个数据结构位于内存中。有内核来进行维护，是每个进程进行上下文切换的一部分。每个进程中都有一个自己的页表来对应这个进程所需要的数据在内存中的虚拟内存中对应的块编号。

这个数据结构是一个被称为页表条目或者PTE的数组。其中PTE[k]保存的是DRAM中物理页面k的物理地址。

需要注意一件事，一个页面在系统中一般具有一种互斥的属性。即，如果这个页面在DRAM上，那么就不会在外存中。如果在外存中，那么就不会在内存中。

这个可以在PTE表中的架构中初现端倪。当PTE表的一个元素中的有效位为1时，代表着之后的一串地址是这个页中在DRAM中，且这段地址是在DRAM中的地址。但是当有效位为0时，有俩种情况，当后面的地址为NULL时，代表着这一段虚拟地址页为未使用的情况。当后面的地址位为一个有效的地址位时，这代表着这虚拟内存页在外存中的实际地址。



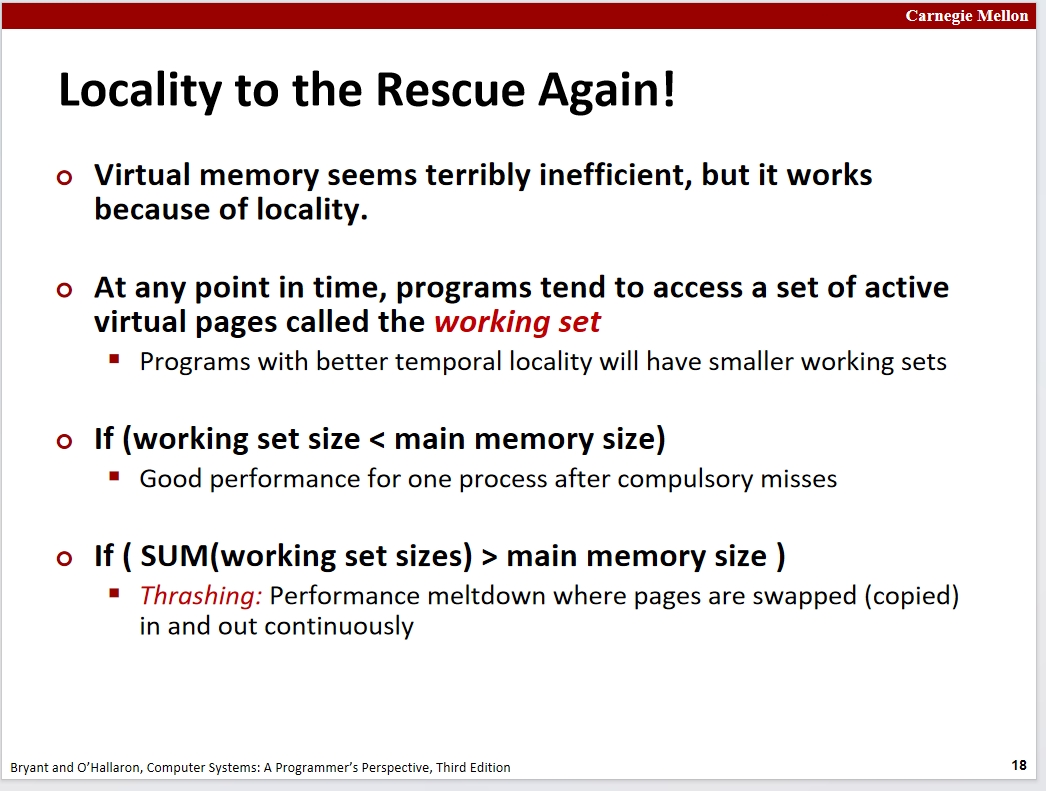
接下来简单抽象一下运行的操作。

首先，CPU先根据一个当前指令抽象出一个虚拟地址，然后，这个虚拟地址会通过MMU转换为页表上的一个索引和一个偏移量，通过对这个页表的查找能够实现数据的查询。有几种情况可能发生，如果有效位为1，则代表着这个对应数据已经被加载到了内存当中。接下来系统将会通过这个页表中的地址数据去再次翻译为一个物理地址的信息。此时这个地址信息代表着主存上的一块数据，系统回去找到这块对应的数据读取并返回。

但是，如果我们在翻译出来的页表中的数据的有效位为0，就代表着这个目标数据想要的页其实并没有被加载到主存中。此时就会触发一种错误，或者说一种异常”**页缺失**”。此时，进程会将控制权交给内核，进入super模式，将我们这个虚拟地址所对应的虚拟内存页复制到主存当中，当然，这需要考虑主存此时是否已经占满，如果已满，则需要进程一定的替换算法进行替换页的选择。通过将替换页的复制到外存中并将目标页复制到主存中。系统就修正了这种错误。此时程序会重新执行这条有关数据的操作语句。

除此之外，简单接触下动态内存分配的情况。当我们使用malloc，new等为一些数据分配内存时，系统会去找到一些虚拟内存中未使用的页，并将这些页的地址信息与具体物理内存的地址进行绑定，毕竟对于这种未使用的页，其的地址中储存的是NULL。但是，一般来说，这些页不会被立即被加载到内存中，应该还是接受一个写时复制的机制。

在刚接触虚拟内存时，其实我确实会因为可能出现的页缺失的现象而感到疑惑。因为这种页缺失的现象所带来的性能损失是很大的。但是，程序的局部性很好的拯救了这个特性。



一般情况下，一个进程其实都只倾向于访问一组称为工作集的页面。就比如你的函数，一般来说使用到全局变量的情况其实不多，大多数的情况下都只会使用到函数内定义的数据和参数等。如果我们这个工作集的大小小于我们主存的大小，那么这个工作集所需要的数据就能够被加载到缓存中，而这种架构能够满足绝大部分函数运行时的需求。但是如果我们进程的工作集的大小比我们的主存大小还大时，这就不是一个好的情况了。这种情况下可能导致一个严重的性能损失。可能会导致频繁的页缺失导致内存的不断循环复制。

接下来看虚拟内存在**内存管理**上的应用

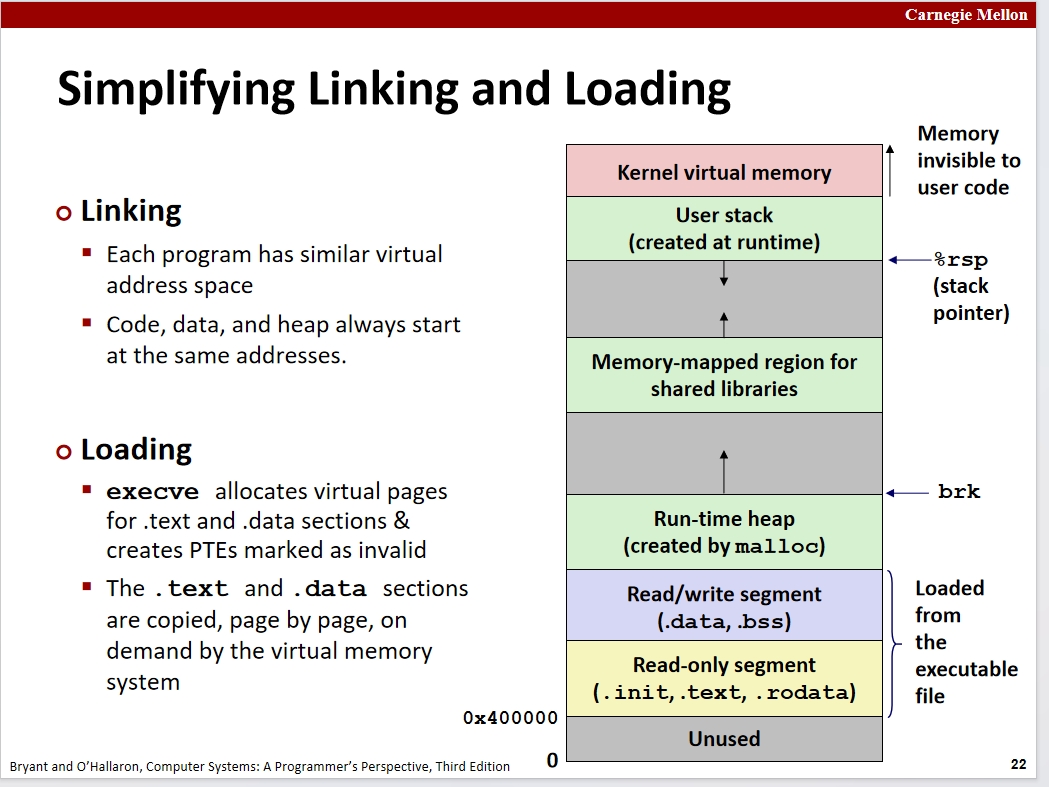
内核通过为每个进程提供自己独立的页表来实现内存管理。这个会位于进程的上下文中，也就是说，这个其实是内核的数据结构。是内核为进程所维护的。

每个进程的页表都映射该进程的虚拟地址。在虚拟地址空间中这些连续的内存地址，在物理内存上大概率其实是不连续的，他们能够被映射到DRAM上的任何位置上。而且不同进程的页表中的表项能够映射到相同的物理页（共享页）或者不同的物理页。

通过这种架构，如果以程序设计的视角，我们可以认为每一个进程具有非常相同的地址空间。具有相同的地址空间大小(因为每个页的大小是相同的，不同的空间大小其实是由于使用页的数量决定)，相同的地址空间架构（认为每个程序的代码，数据，内存区等都是从同一个地址开始的），但其实进程使用的页面很可能是分散在主存中的。但是在程序设计者的角度上不用考虑这些，虚拟内存为我们提供了一个十分有用的视角。

如果没有这种架构，让我们去想象，去跟踪每个进程中使用的数据的位置几乎是不可能的。

虚拟内存有许多十分强大的机制，来看一个好理解的。共享内存空间。虚拟内存运行多个进程的不同页表项指向我们同一块的物理内存地址。这种情况下能够实现一块数据的共享。就比如我们的程序连接，一般c程序都需要链接一个lib.c库。如果每个程序都需要去内存中加载一个内存库到主存中的话，这是一个相当耗费时间与空间的操作。而共享文件的这种特性能够极大的简化这种操作，在这种架构下，我们所有的要使用对应的动态库的程序只需要让自己的条目中包含这个物理地址就能够实现原本需要独立加载的功能。由此可见，使用虚拟内存架构下的系统，可以极大的优化我们链接和加载的过程。



接下来看一下虚拟内存系统在链接和加载上的应用。

首先，正是拥有了这种虚拟地址的架构为每个进程分配了独立的内存空间(就比如常规的从0开始的地址空间)，我们的链接器在链接过程中才能分析出每个程序在运行时对应的数据的位置，即了解每个数据将被加载到哪里。因此，链接器可以解析所有编译器给他留下的重定位信息。

用execve的加载进行下程序加载的举例，当进程运行execve时，内核会去查看这个execve想要加载的数据段和只读文本段中的代码等数据，并通过一系列算法分析出这个代码到底需要多大的内存空间，并给他预备一些可能需要到的内存页。但是吧，在加载的时候，其实并不会将所有数据即刻加载到内存中，在加载时，内核为这个进程创建PTE表，但是PTE表上的很多页表项的有效位为0，这个会导致在需要对应的数据时进程再通知内核去进行对应页的数据往内存上的复制。也就是说，这个就是一个写时复制的实现原理(简化的不能再简化了)。

在此处我对一个运行的程序有了更好的理解，再来分析下吧。

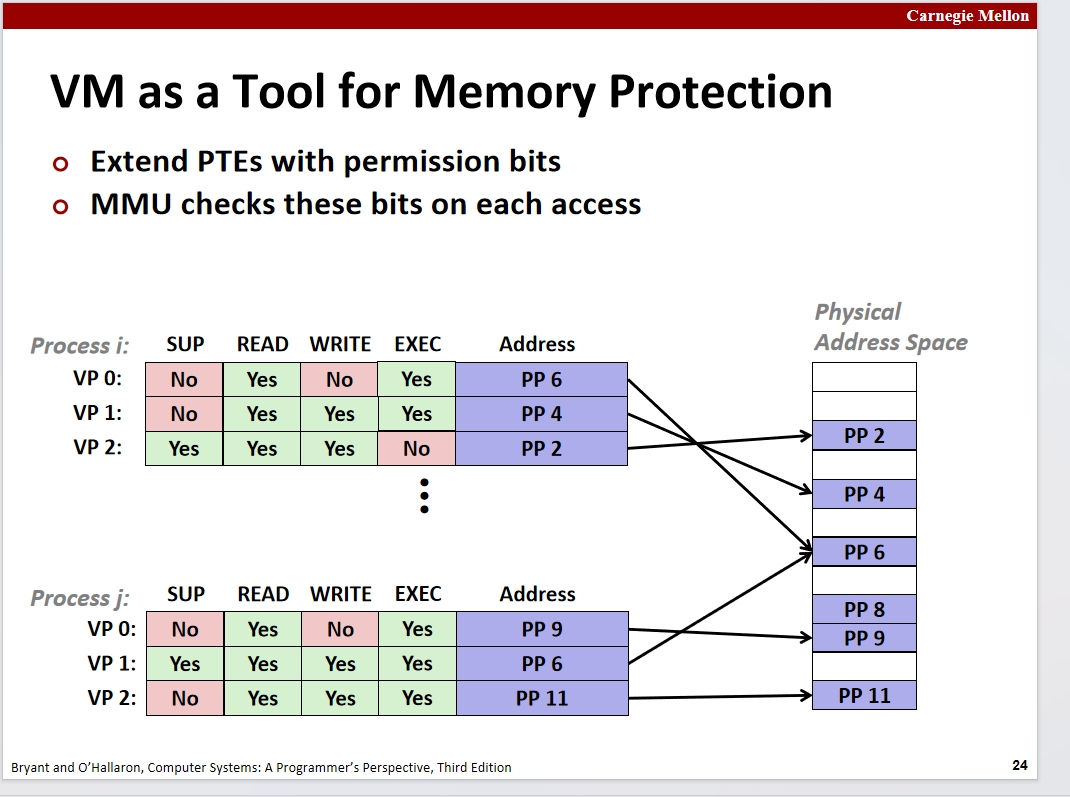
我们的一个所谓的程序的运行，其实就是一个可执行文件的代码段的不断读取和对应的操作执行。在一开始的时候，这些可执行文件是储存在外存中的。

当我们想要执行一个可执行文件时，我们需要一个触发来进行我们代码的执行。这里就需要一个execve的系统调用。当进行这个系统调用的时候，系统会读取对应的可执行文件，并识别对应的只读文本段等并给整个可执行文件分配一定的虚拟内存页。

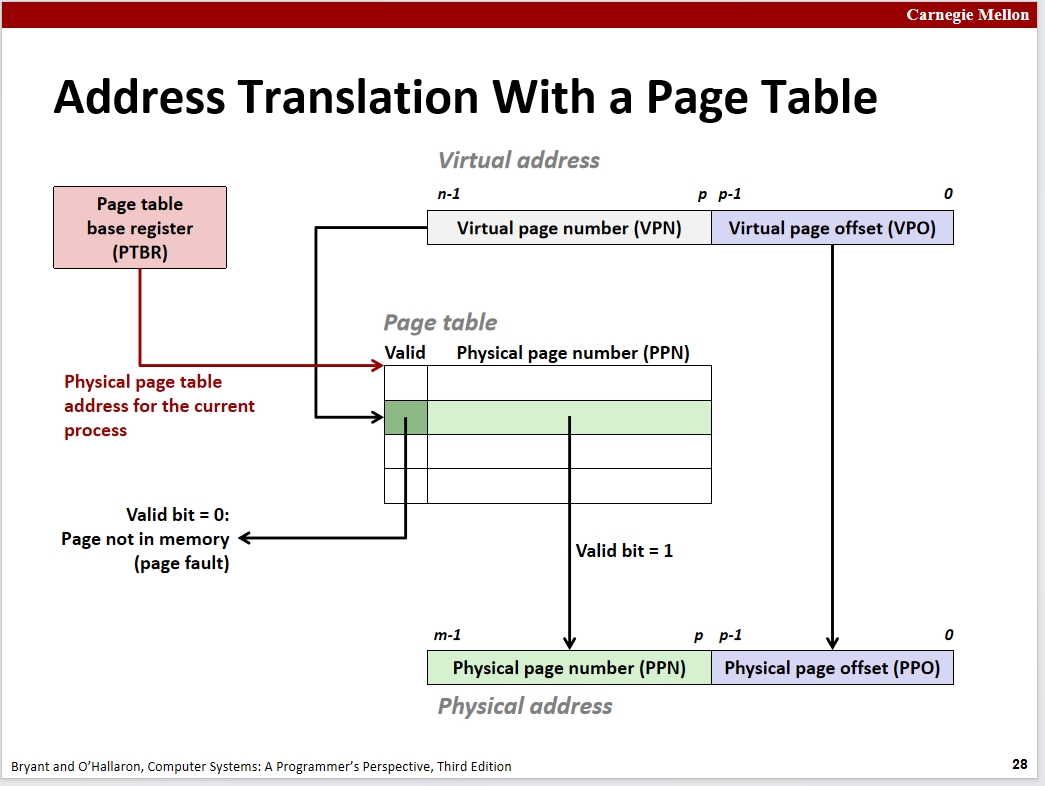
在读取的过程中，内核会识别这个可执行文件的代码入口，就比如main函数等，这个通常是在文件的头部表中的。接着，内核会将这个入口位置的地址送到一个CPU的PC当中。与此同时，内核会初始化一系列的进程需要的东西，这里就考虑PTE表。

在这个表中，包含了这个可执行文件的各个文件的具体数据的映射，初始化的时候大部分的页表项的有效位都为0，表示都没有被写入到内存中。但是有些例外，就比如我们包含了只读文本段的代码的页。这个页通常会在内核将入口地址送给PC时从外存中复制到主存中，此时这个在页表对应的页表项将被置为1.接下来，对于其他的数据页，则是会通过写时复制再加载到内存中了。

由于只读文本段中的代码等数据是不可能被修改的，也就是不可能会触发写回等操作的，所以这种预加载的架构能够更好的提高效率。



接下来分析下虚拟内存系统在内存保护上的运用。对于每个页表项，可能不止一个有效位，还有一些可能的权限位。就比如代表是否能够被当前进程读写的WRITE和READ位，代表权限(内核访问或者进程访问)位的SUP等，通过添加这些位，并且每个进程上的页表中的这些位上的数据并不一定相同的情况，能够使得我们进行不同进程的对同一个文件的不同访问权限。



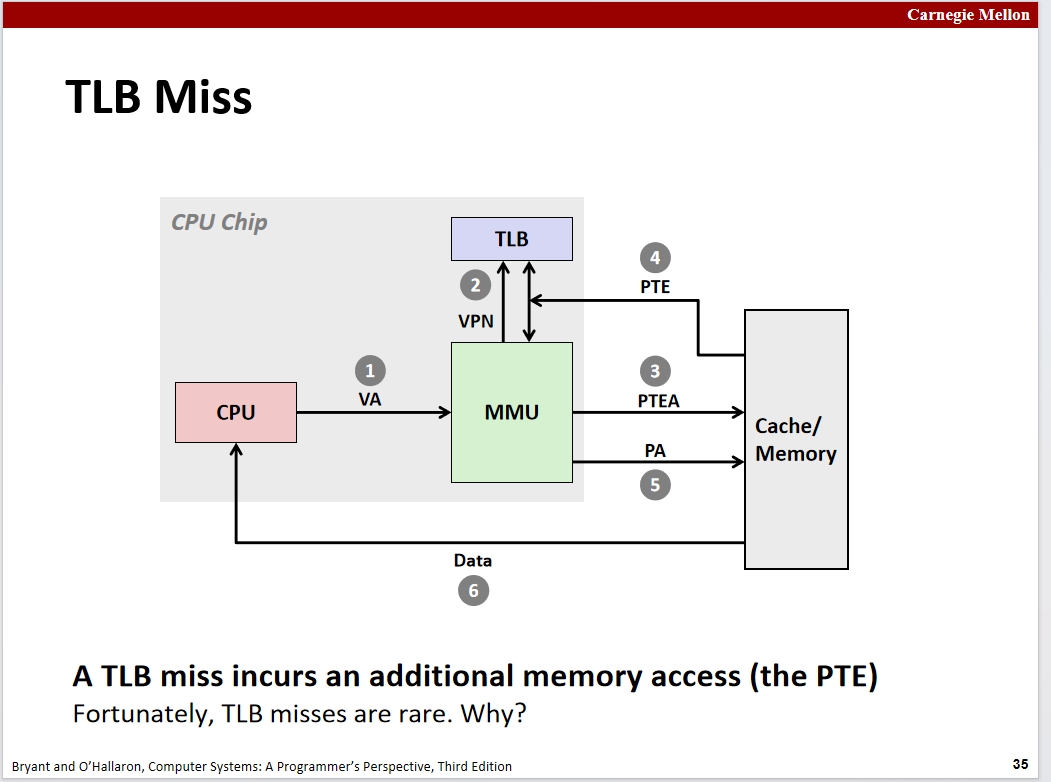
接下来让我们分析下虚拟地址的运作方式。首先，CPU会根据代码中的执行指令将对应的虚拟地址发送出去，前面我们了解到这段地址会被送到主存中去，但是在这里我们再进行细化。

在虚拟地址被发送后，其会先被发送到CPU中的一个MMU中进行虚拟地址的翻译，是的，对于CPU来说，由它发射出来的地址都是一个虚拟地址，我们是没办法通过这个去找到实际的数据的。MMU通过对这个地址进行解析，通常一个虚拟地址包括俩部分，一块是前面的虚拟页(**VPN**),这个是用来进行进程页表的索引的，还有一个后面的偏移量(**VPO**)，这个是用来具体的定位的，当有了这个后，我们就可以去找到对应的条目表中对应的数据，而这个数据一般也就是我们想要的数据对应的物理地址。

物理地址的架构与虚拟地址的架构极其相似，都是有着内存页编号(**PPN**)和偏移量(**PPO**)，而且这个偏移量的位数一般也是一致的，位数区别主要是在前面的页编号。

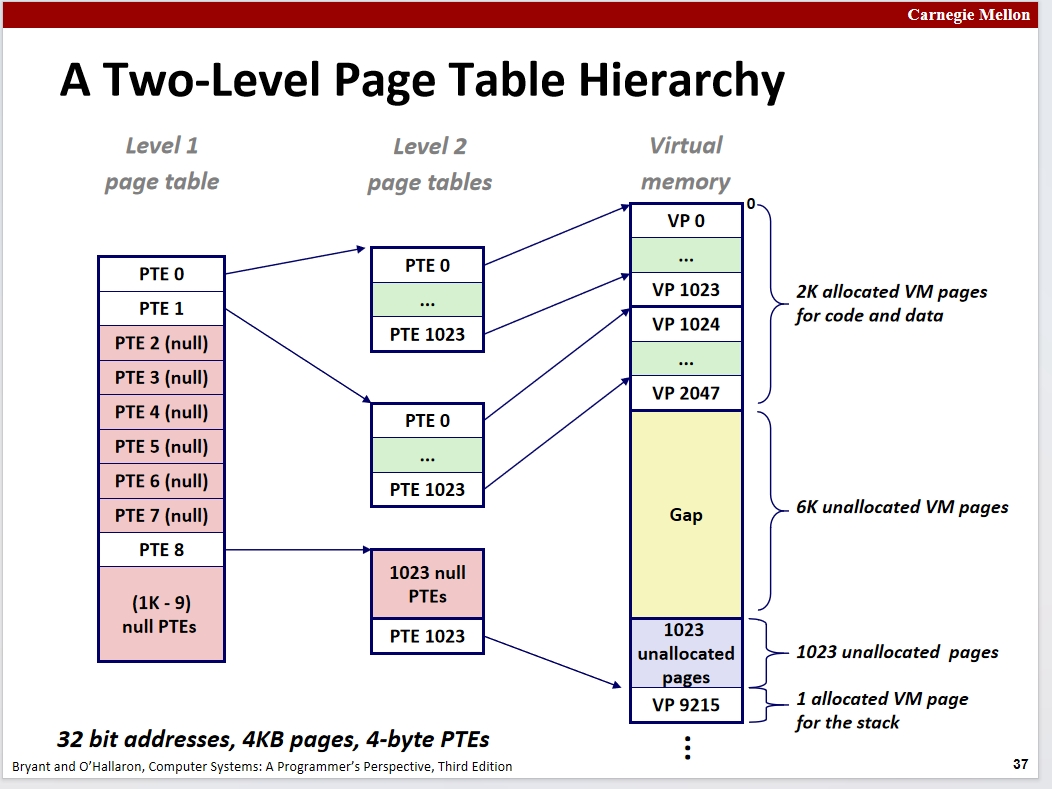
当MMU通过上面的翻译后就能够取的真正的数据所在的内存地址了。接下来MMU会将这个地址继续通过总线发送给主存，然后主存将会返回这个对应的数据。

当MMU在解析时遇到符号位为0的情况时，进程会发送页缺失的异常给内核并进行对应的异常处理。处理完成后会再次进行一次MMU的解析操作来取得数据。



接下来分析下MMU的缓存规则，由于一个页表条目可能会很多，为了加速这个访问过程。系统为MMU也设置了一个用于缓存的TLB，这个设备用于储存最近访问到的页表条目，如其他缓存一般，当CPU发送一个虚拟地址给MMU时，会先去查看MMU中的TLB是否已经缓存了这个条目，如果已经缓存，直接通过这个来进行对于物理内存的构建，其他行为并无不同。未命中时的情况也一致，当未命中时，系统将会直接通过MMU进行操作，并在之后将这个条目缓存到TLB中。

对于页表的架构，如果使用直接一一映射的方式来构建页表，页表的大小将会是不可接受的。所以这里就引入了页表的层次架构。



每一个块由于位的大小能够表示映射到不同的位置上，就比如11位的01串，就能够映射出2^10-1个数字，也就能够映射出对应多的块，虽然实际上是否会映射出这么多的块成疑。对于每一个字块，又能够映射出对应大小的字块。一般来说，这个映射的层次结构有4层。

对于这些页表项，为空时会代表着这一块是不会映射出对应的子块的。这些块就可能是系统内核为一个程序预留的堆栈区。这些内存区在未被使用时不会被分配条目。这样就能够简化搜索的步骤并且对于系统的层次更加清晰。

对于系统给一个进程分配的内存的大小，其实并不是一个确定的值。对于一个架构，理论上是能够全部填满的，但这也只是理论上。在实际操作中，这一个一级架构中一般为0的表项还是占多数，即使在为1的表项的子表项中，为0的也还是占多数。这里就需要注意一个情况，就是说对于一个虚拟内存页，如果它里面并没有数据，或者说这个虚拟内存页没被使用。那么系统是不会分配物理内存页给它的。这种设计能够优化这种架构的属性。毕竟你如果想要为每个无论使用与否虚拟进程页都分配一个具体的物理内存页，这个开销也是不可接受的。使用这种按需分配的设计能够极大的平衡设计与使用。

同时，如前面所说，一个物理内存页时能够被多个虚拟内存页所使用的。但是，这里我们想要说的不是这个，我们想要说的是这种架构下，我们能够更好的去利用外存中可能被分割的内存块以实现更好的利用率。各个虚拟内存页对应的物理内存页的物理位置是不能确定的，这种不连续性能够更好的利用储存空间。

正如上面那个架构表，对于一个进程，若其的条目表的有效位为0，就意味着这事一块没被使用的内存块，对应的虚拟内存页页不会被分配。以此类推来进行内存上的优化。