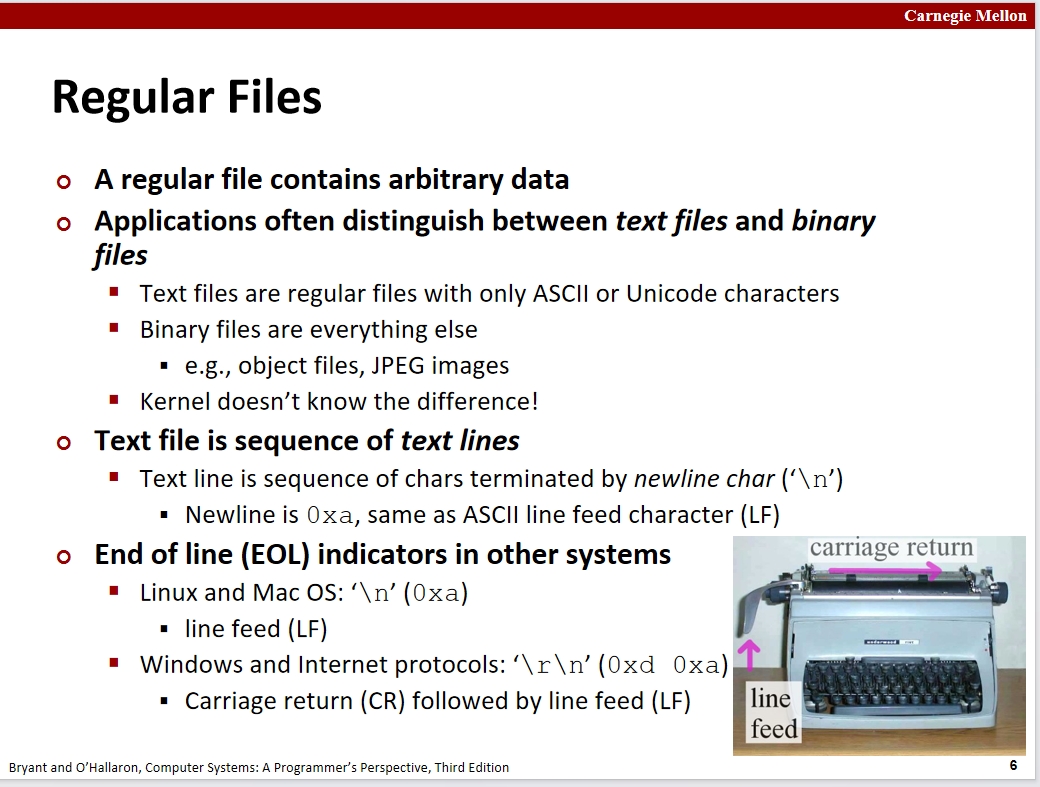
对于文件，可以看做是对于各种层次的抽象，对于内部软硬件的抽象，对于外部软硬件的抽象，对于网络通信的抽象等等。

这里只考虑几种常规的文件:  
 

**普通文件**

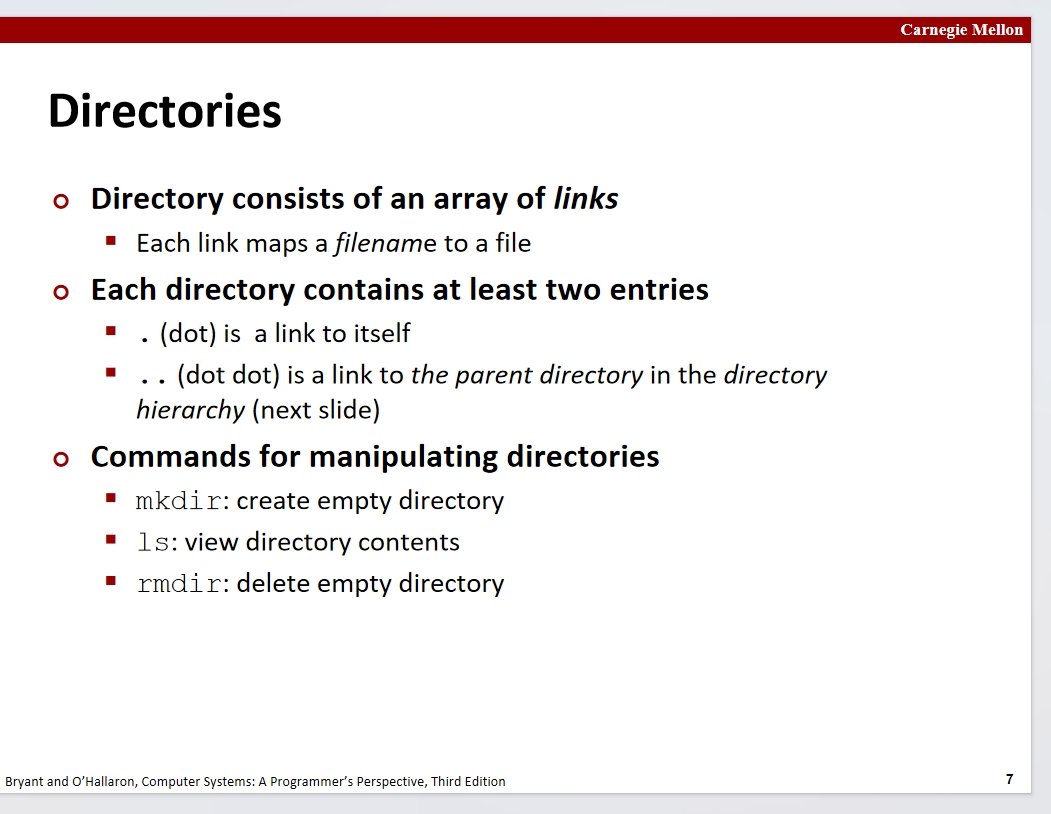
对于一个普通文件，操作系统不会去尝试探索文件中的具体实现细节。

有些程序可能回去区分这个文件是一个普通的文本文件或者是一个二进制文件。对于文本文件，其中储存的是标准的英语的ASCII字符或者代表非英文字符的Unicode编码等。

一个二进制文件通常是图像文件或实际目标代码或音频文件等所有的其他类型的实现类型。

在文本文件中，这个文件包含的一些函数被视为它的特征。这些函数能够识别这个文本文件的换行符，也就是一行文本的末尾。指的是这个文件中数据的一个换行，而不会是对应的文本抽象出来的数据的数据的换行。对于一个文本文件，其本质上其实不会是屏幕上输出的一行行的数据，而只会是一个连续的流，这一个流中储存的是整个文件的数据。

当一个文件在不同系统中的转移时，需要特别注意这个换行符的识别。在Linux和Mac OS系统中，对于文本文件中换行符的规定是将’\n’(0xa)(LF)识别为一个换行符，但是在win系统和网络通信中，对于换行符的规定是’\r\n’（0xd0xa）(CF)(一个回车一个换行)，故当一个跨系统的文件进行传输时，如果对这个换行符不进行任何的修改，那么在传输的时候会导致最终的文本识别出现乱码



**目录**

目录跟前面的普通文件在操作系统中的储存方式是一致的。但是，通过一个与普通文件不同的解读方式(或许在操作系统又或许在操作系统的文件系统中)来进行一个区分。

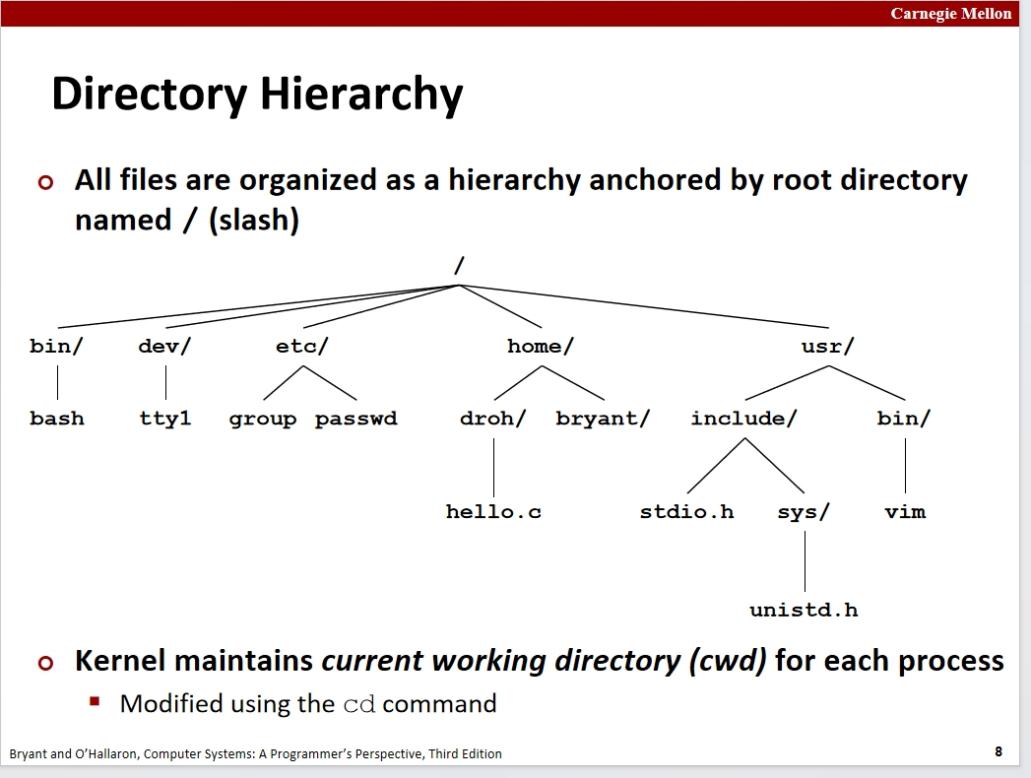
对于目录，包含俩个特殊的条目(.和..)分别表示当前目录自身和树形结构中当前目录的父目录

例如，如果当前路径是 /home/user/documents，那么在 documents 目录中，./file.txt 就指向 /home/user/documents/file.txt

.. 就指向 /home/user

对于这种文件路径，最后的一块是代表着当前位于这个文件目录本身

对于文件路径的使用，如果在路径前面加上/,这意味着使用的是绝对路径，使用这种开头的路径会沿着层次结构顺序向下查找

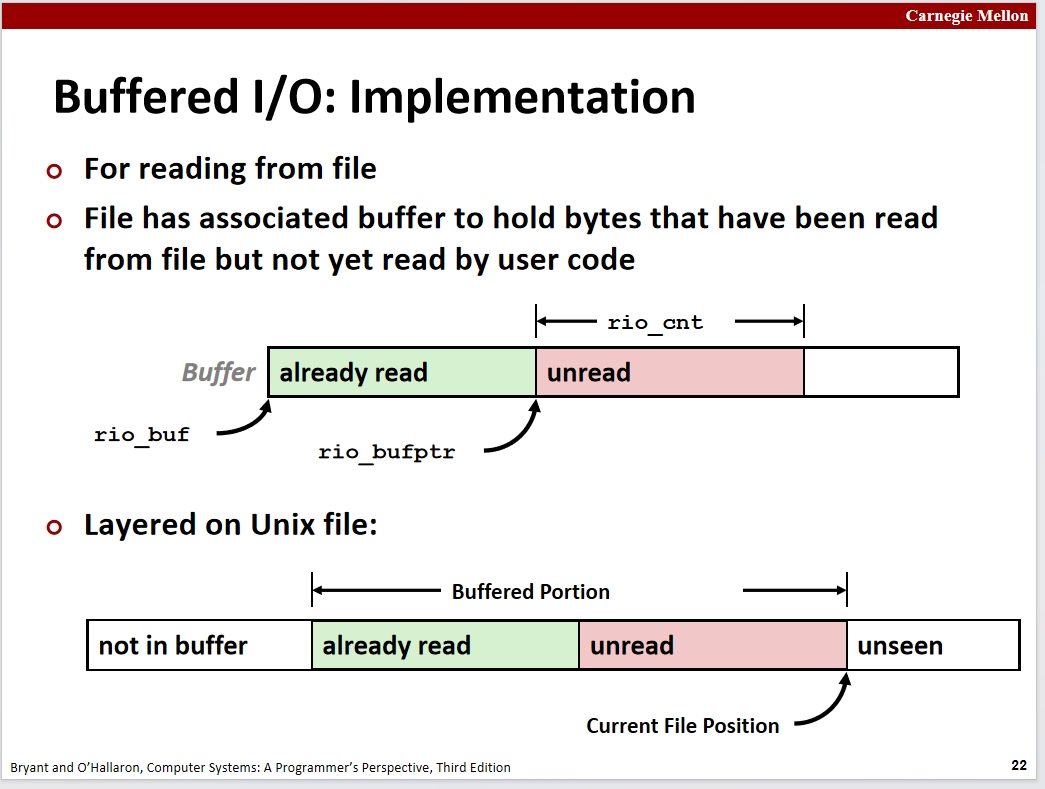


当然，也可以通过.和..开头的路径来实现相对路径的构造。需要注意的是，这里的相对路径其实是一种隐式的绝对路径，只是这个隐去了当前目录前的部分而直接由.来替代

在进入接下来的内容时，我们需要先了解一个重要的概念:  
**文本描述符**，这个文本描述符是操作系统用来标识使用的文件的。对于每个进程来说，都有一套自己的文本标识符，最基础的几个文件包括：标准输入文件stdin(标识符为0),标准输出文件stdout(标识符为1),标准错误文件stderror(标识符为2)，除此之外，在一个进程中打开的文件会依次在这三个文件之后进行标识符的分配。对于程序，其是直接通过文件标识符来对文件进行使用的，包括但不限于一系列的读写操作。

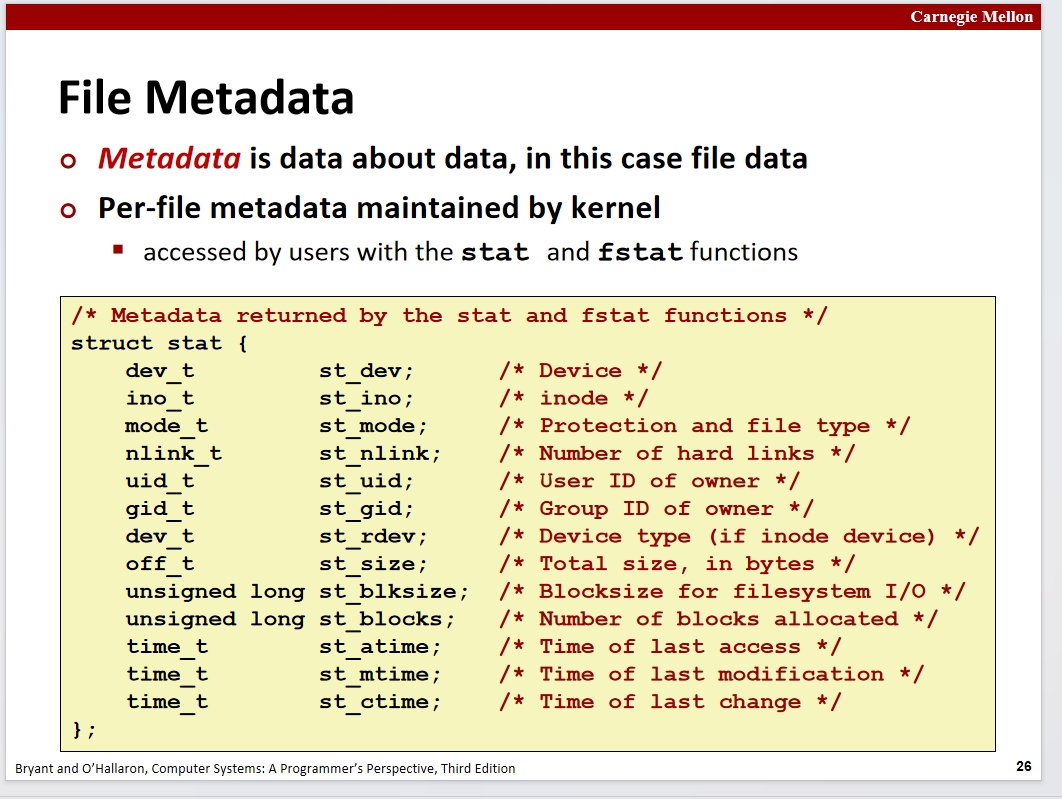
当我们打开一个文件时，我们其实是把一个文件跟一个操作系统给内核分配的文件标识符给挂到了一起。之后的文件操作操作系统通过这个文件标识符来对对应的文件进行操作。

文件的访问等一系列操作都是一些系统调用。所以正确的编程规范是对这些涉及到系统调用的操作都应该检查其的返回值(一般都有且对应了一定的错误信息)。即使是关闭文件这种操作，也应该查看对应的返回值。毕竟关闭文件这个操作也涉及到了内核的系统调用，如果在调用过程中被信号所打断而我们却不加以检查的话，之后的所有关于文件的操作都是无法实现的，而且是难以排查的错误，毕竟你也许在下次运行时就不会发生这种错误了。



接下来来看一种能极大提高程序读取数据效率的工具：带缓存的I/O操作。这种工具一般使用与大规模的数据读取，但是程序却需要分多次操作的情况。由于读取数据需要涉及到系统内核的调用，涉及到上下页的切换，所以我们其实是希望尽量减少系统调用的频率和次数的。这里一种缓存的思想就能实现对应的操作。

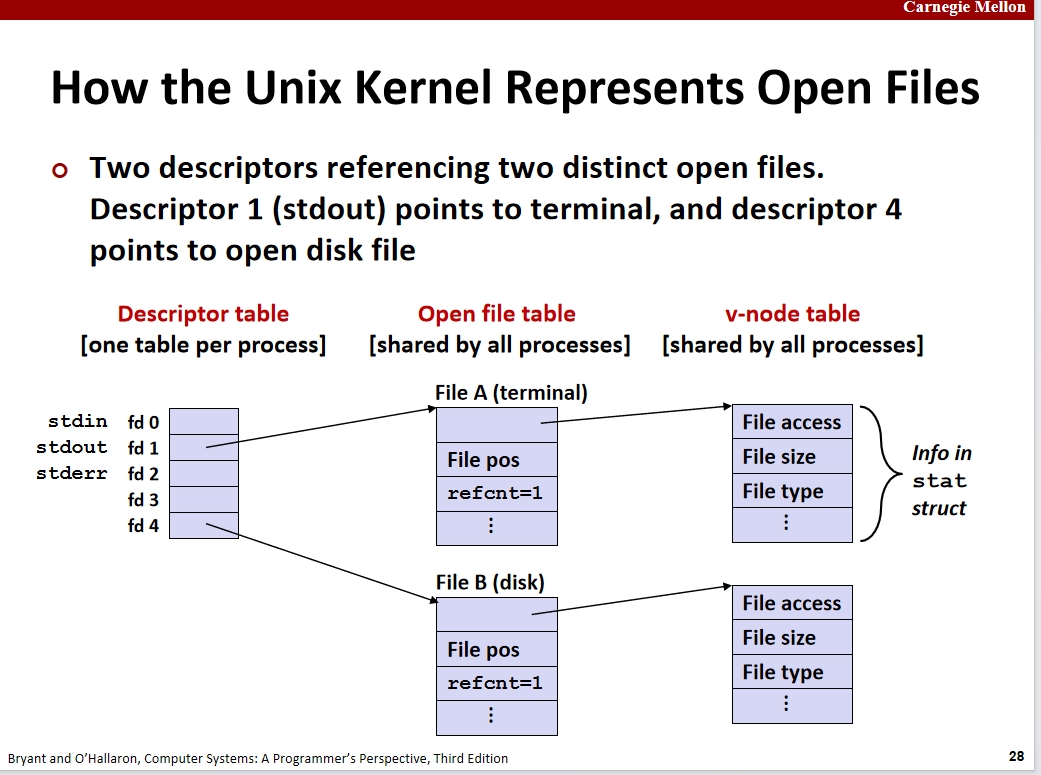
对于带缓存的输入输出等，每一个操作对象都带有一个对应的缓存区，就比如书中的那个给read函数绑定的结构体缓存区。在这个缓存区中，一般存在一个字符数据来实现对于数据的暂存，当存储数据的数组不为空时，程序会读取对应的数据到指定的位置上并将读取位置(指向这个数组的位置进行更新，一般都是直接更新到起始位置)。然后程序再一次性的去读一块内存数据。通常来讲，一个缓存区中的数据能够满足多次正常使用。这样就减少了一定程度上的系统调用



接下来看一下文件中的其他部分

一个文件中会包含一系列的信息。这些数据被称之为**元数据**。包括但不限于这个文件的读写权限，文件创建时间，最后一次访问时间，最后一次修改时间等

想要获取这些数据，可以使用unix等系统存在的stat函数，通过给其传递一个目标文件的地址作为第一个参数，一个stat结构体(系统内置结构体)地址作为第二个参数，就可以将目标文件的这些元数据拷贝到这个结构体中，对应的数据与元数据中的名称相同。



接下来看一下在进程中对于文件的管理。

对于一个进程，其会维护一个Descriptor table,这个表中储存了这个进程中所有使用的文件所对应的文件标识符。对于每个文件标识符，其有对应着一个内核管理着的一个文件。或者说，这个文件表表项的每一个都是对某个打开文件的引用。这也是为什么我们的read等文件操作函数只需要一个整形数字就能实现对对应文件的操作。

同时，这个表的表项中还储存着一些信息，包括这个文件在系统中的具体位置，这个文件最后一次读写操作的位置等。除此之外，还有一种操作系统用来追踪文件是否正在被使用的引用计数器。由于使用一个表项，也就是打开一个文件，这个操作可能是有多个进程共享的，这个引用计数被操作系统用来进行追踪内存分配，使得让操作系统知道在什么时候不需要这个表项。就比如文件在一个进程中被关闭导致文件不可访问等。

请注意，这个引用计数计的是对于这个文件当前使用的进程数，内核没检测到一个使用该文件的进程，会在所有使用这个文件的进程的文件表中的对应表项含有的计数器加1.对于一个文件，当且仅当这个计数器归0时，程序才有可能将这个文件关闭。不然就可能出现在一个进程中关闭了一个文件，但是另外的一个进程中还是在使用这个文件，这种不同步会导致一种错误

当我们在程序中调用 close() 函数时，它并不意味着文件就会立即被系统关闭。如果其他进程仍在使用该文件（即引用计数 > 0），那么文件就不会被真正关闭，而是仍然保持打开状态，直到所有进程都关闭该文件。

接下来进入到第二块（Open file table）

这一块没什么好说的，这些就是系统上的文件，由所有进程所共享。主要看这些文件中的一个部分(v-node table)，这些是文件的虚拟节点。这些节点也包含了一定的信息。包括但不限于每个文件stat中的信息，文件在系统上的储存位置，文件的大小等。这个表项无论文件是否打开都会存在在文件中。

接下来看一种有趣的情况，如果你在一个进程中多次打开同一个文件又会怎么样呢。首先，这种操作是合法的。内核会会为这每一个的打开操作都分配一个文件描述符，或者说，在表中分配一个条目。这个条目中一定包含着一个偏移量信息，这个信息如之前所说包含着最近一次对文件的读写操作的位置，那么这俩个描述符就包含着俩个偏移量信息。也就是说，如果我们的这俩个偏移量不同，我们就能够实现对于一个文件在不同位置上开始读写操作。

接下来分析下多进程对文件的影响。当我们在一个程序中fork一个子进程来使用的时候，这个进程复制的数据会包括父进程的文件条目表，也就是说这个子进程会包含所有父进程对文件的操作信息，只有在任何一个进程中对文件进行操作时，这个信息才会逐渐产生差异。需要注意的是，各个条目项中的对应的程序计数器会在fork之后自动+1.