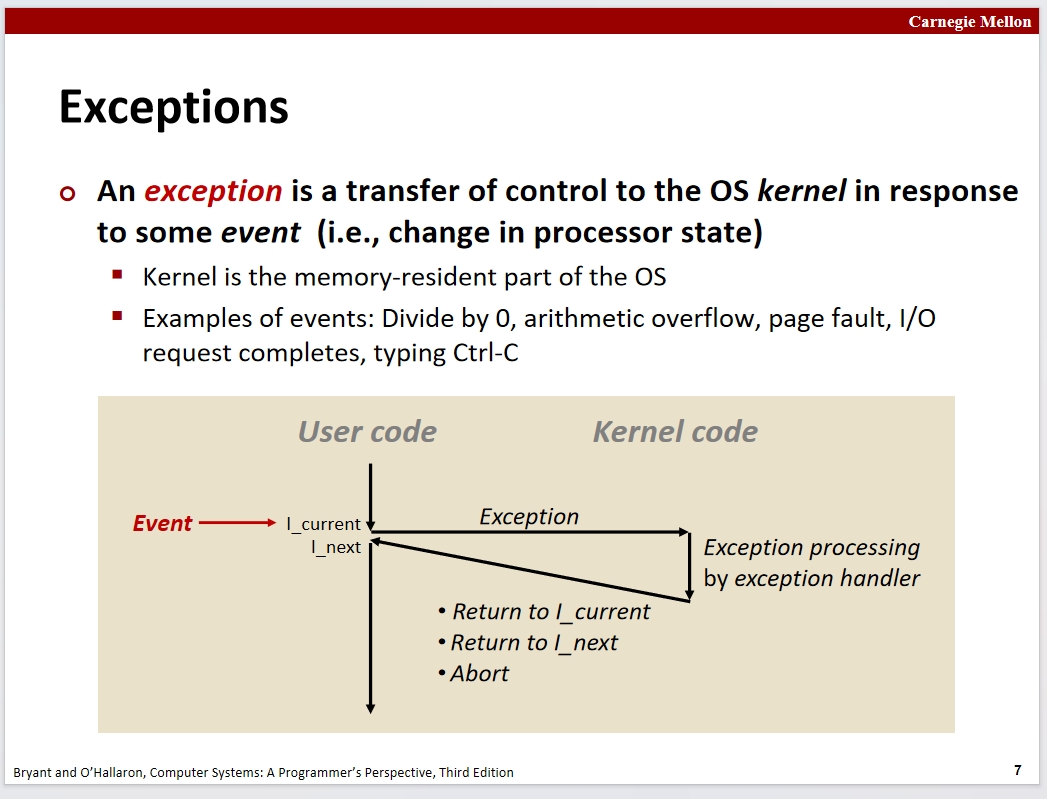
**异常控制流(遍布操作系统的多个层次)**

物理控制流：硬件正在执行的实际指令序列。一般情况下，该控制流在内存中按顺序执行单条指令

首先，我们需要对异常进行一些了解:

异常归根到底，还是一系列的状态而已，就比如状态机中的一些特定状态，我们正常情况下运行的代码对应的也是一系列状态。他们不被视为异常的原因是因为这种情况是与我们的预期不相符或者这种情况是会导致一些其他的问题。

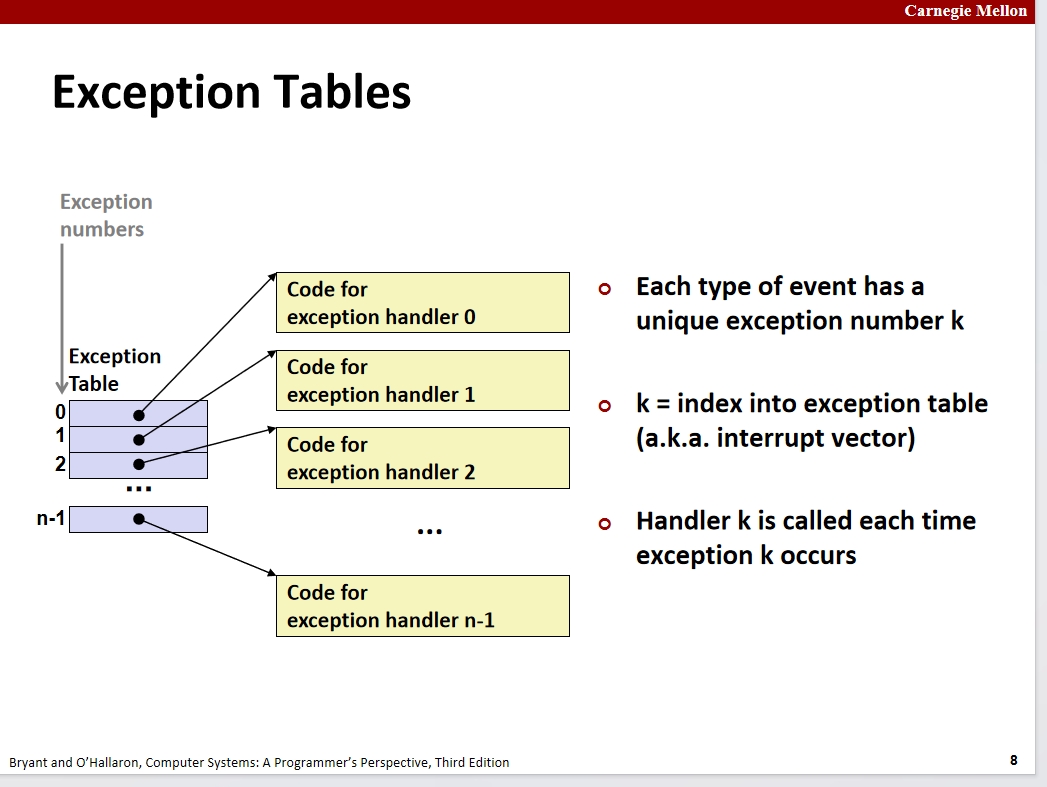


当遭遇异常时，程序会将控制权转移到操作系统在内存中驻留的内核中，如果是以状态机的视角来看，这种情况下就对应着这些特定的被视为异常的状态将转移到内存中进行异常处理的操作系统内核的状态。接下来，这些处理程序将会通过规定的代码所代表的一系列状态进行转移，在着进行异常处理后可能会导致三种处理方式，或者说，这个可能导致状态机转移到接下来的几个状态

1~返回出现异常的地方并且重新执行当前指令，这种对于[页缺失]

2~放回要执行的代码并跳过当前出现异常的代码并执行下一行

3~退出当前程序[中止]



异常由硬件和软件共同实现，出现异常时的程序控制权的转移是由rip和其他一些硬件实现的。但是，对于出现情况是的异常处理这一块的代码是由操作系统内核设置和确定的。

对于每种异常事件，都有一种异常事件的编号，你可以将这个理解为应该给规定的部分情况定义为了异常。对于这些编号，它将作为一个**跳转表**的索引，这个跳转表可以称之为**异常表**，也就是是说，程序会在每一步时都进行一个异常表的维护，会检测这一步的状态是否对应的相应的异常(异常表中的索引)，

对于异常处理的情况进行分析下，在程序运行的时候，程序是不会对全部数据进行监控的。相反，它只会监控一些特殊的区域，就比如，当一个操作出现越界时，一些特定的标志位就会发生改变，而程序会时刻监控这些特殊的为，当其发生改变时会触发一些信息，使得操作系统和硬件等进行对应的状态转移。所以，这可以看做是一种局部的监控和触发的操作，异常识别不是全局监控的。当然，对于这些标志位的检测，其实也可以看做一种全局的监控，如果想到计组的计算机架构也不难理解，这样设计使得距离近的可以更快的响应，获得更高的效率，使得不会因为一些低效的操作而导致一些错误

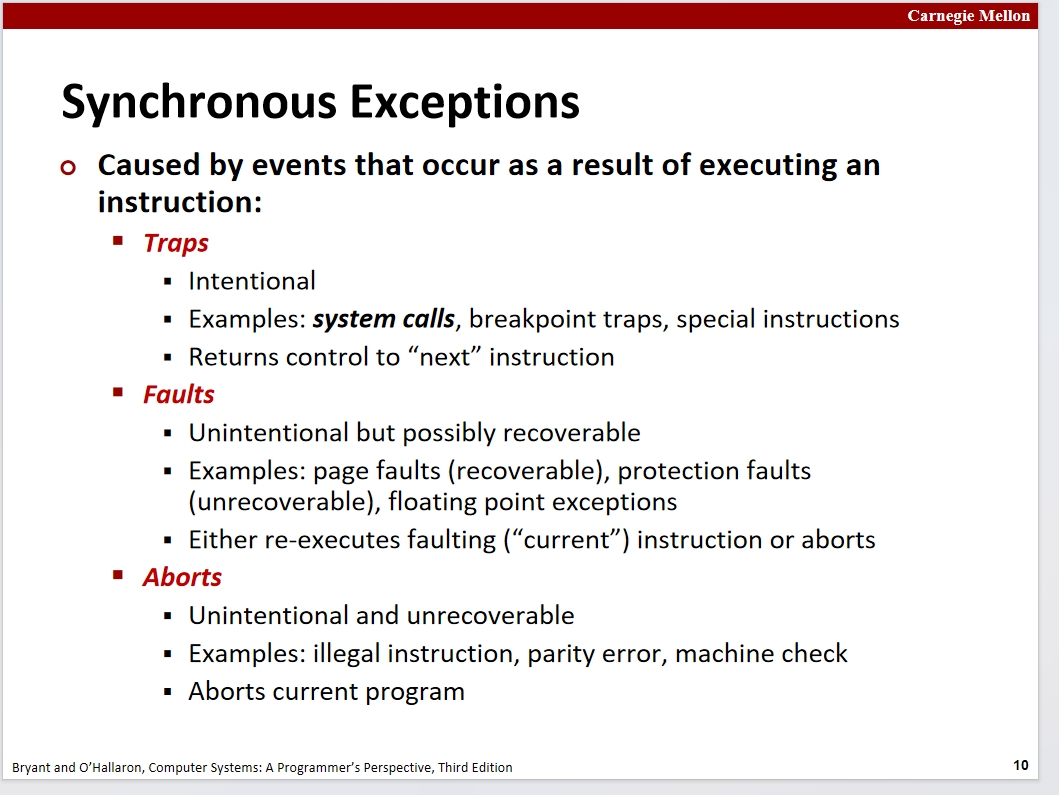
**异常处理的分类（异步，同步）**

异步异常通常是由处理器外部发生的变化而引起的，这种一般称为**中断**，这种异常通常是在处理器上设置引脚，向处理器通知这些状态变化，外部引脚称为中断引脚。这个引脚就是之前磁盘控制器访问主存使用的引脚。

需要注意的是，对于异步异常，通常都是系统设定而实现的，毕竟正如前面所说，异步异常一般都要使用到引脚，这样就意味着它的效率其实是更高的，将其作为一些系统设计会更提高效率。

就比如计时器异常，这个异常是一个计时器异常，一般每隔几毫秒，这个异常就会被触发，此时系统的控制权就会交给操作系统，接着再由操作系统接着决定系统接下来要实现什么，要接着执行接下来的代码还是什么，就比如对于一个死循环，如果没有这个异常，系统将会无法取回当前的控制权，将会导致一些意料之外的后果。

在计时器异常下，操作系统将能够实现不同进程的调度，



**同步异常**

对于同步异常的分类:  
1~陷阱(Traps)

程序故意引起的异常，最常见的就是[**系统调用**]，就如上文提到的计时器异常的要求相似。已知，操作系统提供了一系列有用的函数，当时我们当前自己的程序是没有权利调用这些函数的，就比如对文件进行读写，增删，内存分配等。这些操作是操作系统的特权，其他的是没有权利进行这些操作的这些函数在我们先前所讲的内核中，而这种就是[陷阱]就是将先提供一些函数调用的信号，接下来，进行同步异常中断，接下来将程序的控制权转移到内存中的操作系统内核中，这个内核将会根据我们提供的函数操作信号调用对应的api，执行对应的操作

这种异常将会返回至产生异常的下一条语句进行执行

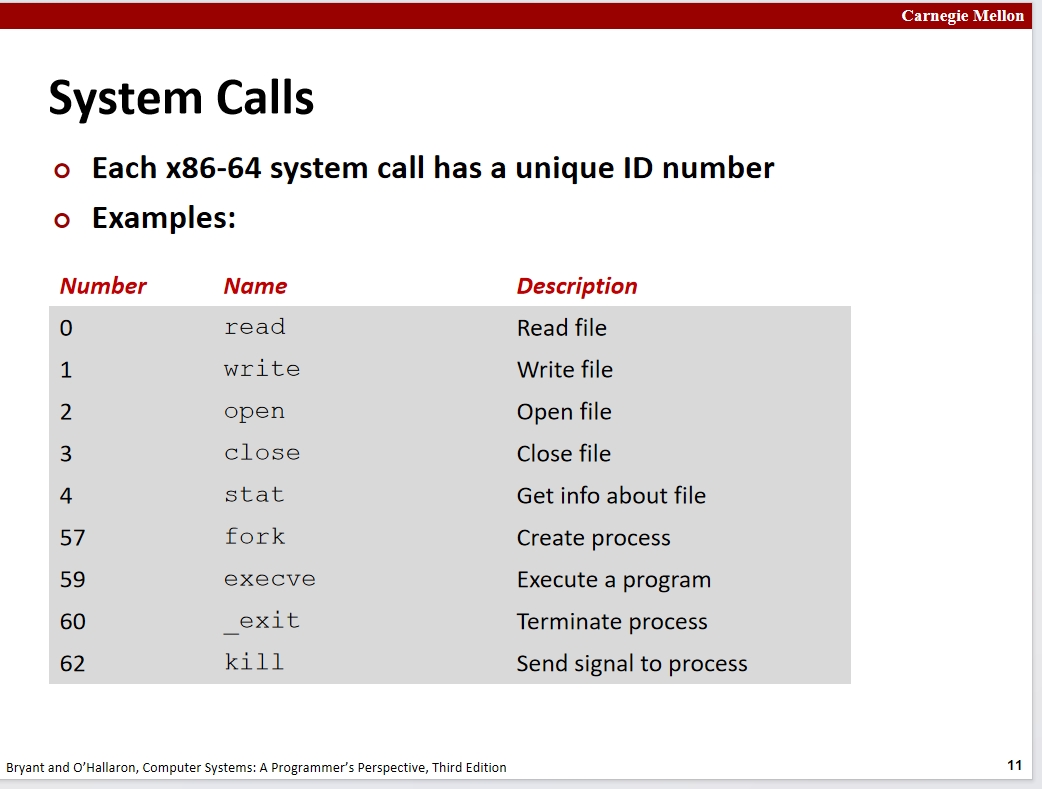
2~故障(Faults)

因为一些系统原因导致的故障，这个其实也分为俩种，一种可恢复一种不可恢复。就比如页缺失的故障。首先回顾下页缺失，页缺失是指程序在访问主存RAM是，所访问的主存并不存在，这种一般指程序引用的一部分地址空间，数据部分不在操作系统给这个程序分配的储存空间上，程序需要系统重新提供这些数据。此时，程序会触发异常，接下来，控制权交往内核中，内核将我们需要的内存复制到我们当前程序所分配的页中，如何内核会重新返回这个调用位置，重新执行这个语句，这是页缺失的错误应该就被弥补了。

还有一些不可恢复的故障，就比如保护故障。段错误就是其中一种，当程序尝试访问未分配的内存空间，这通常意味着程序的代码出现问题，需要对代码的逻辑进行修改，而这个是操作系统的内核所无法处理的，所以这个是一个不可恢复的典型故障。

3~中止(Aborts)

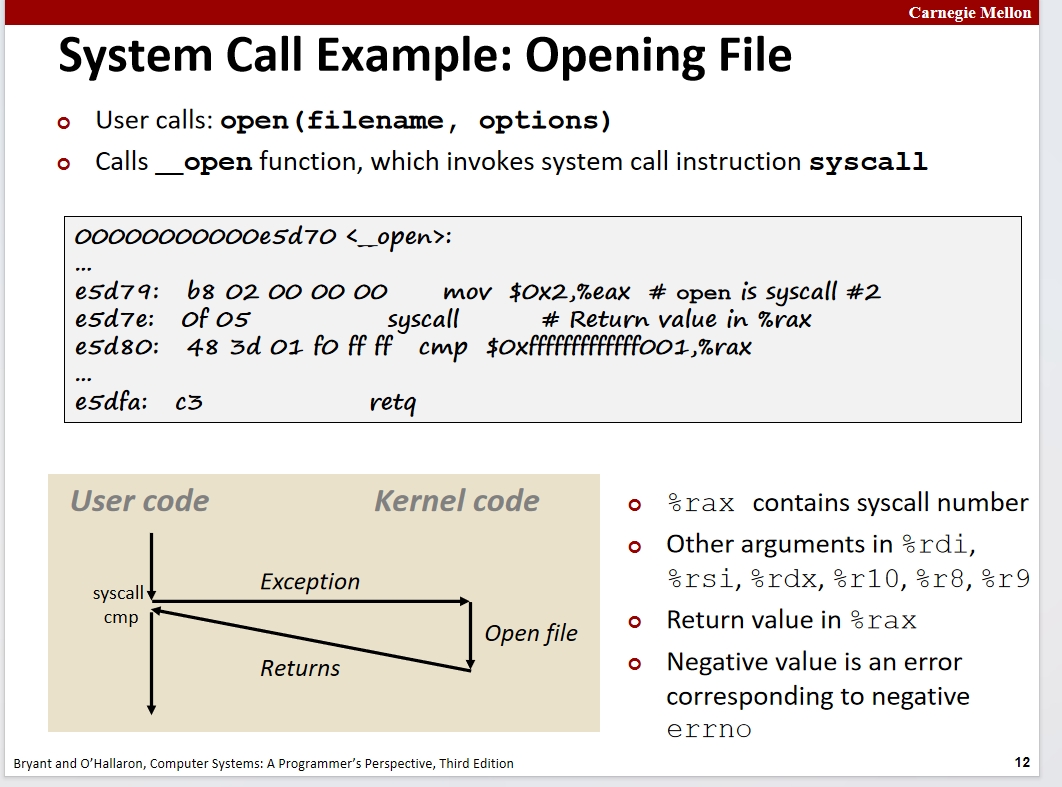
这种异常一般是由硬件故障或严重错误引起的，在这种情况下，操作系统无法恢复程序的正常执行。因此，操作系统会中止当前进程的执行。



操作系统的内核中的函数都有各自对应的编号

上图是一些内核函数中对应的编号

对于汇编，有一个syscall指令可以实现系统调用(虽然一般是不用 的就是了)一般都会自动给你调用的

对应syscall指令，其会使用特定寄存器作为其的参数，如图

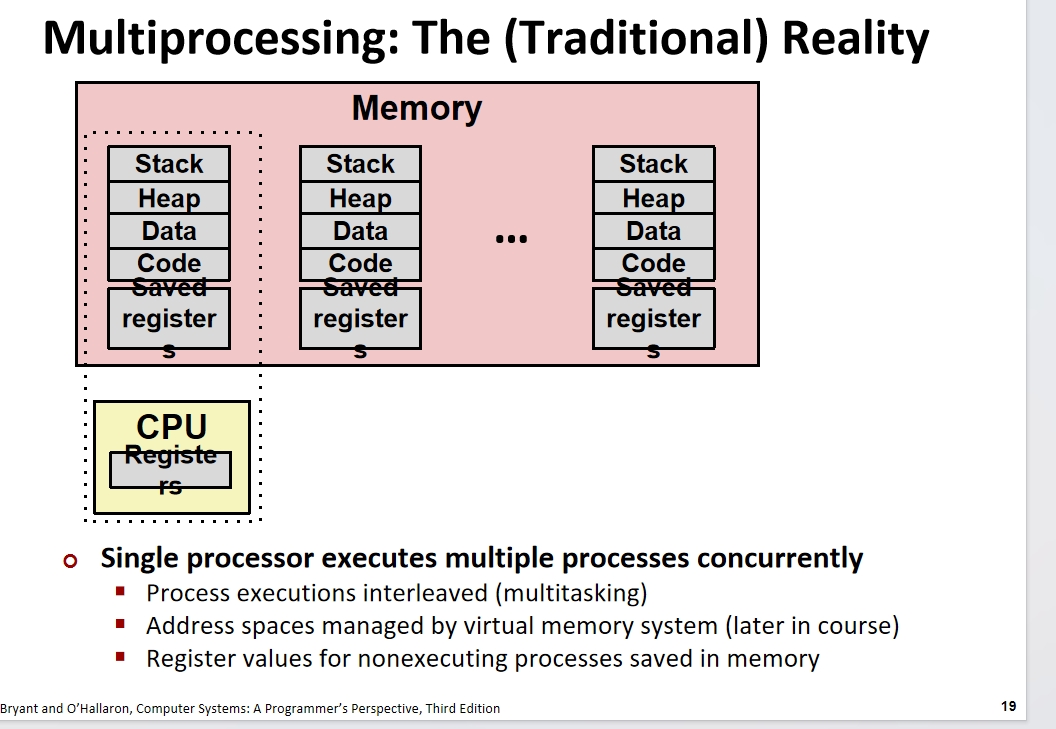
对于这个指令的返回值寄存器%rax，如果它的值为一个负数，这个就通常意味着发生了一些错误，正数一般意味着正确

接下来来接触一个相当重要的概念：  
 **进程**

对于**进程**的定义一般是：当前正在运行的程序实例

对于进程，可能给人的幻觉一个就是你独占了整个处理器，就比如你会感觉在这个阶段你管理了所有的寄存器并不允许额外的写入。还有一个幻觉就是你拥有了自己的一块地址空间。这个说是有一块虚拟地址空间提供的。对于一个程序，你拥有自己的一个栈，堆，代码，数据，并永远不会看到其他程序的内存空间。

但是，实际上，现代的CPU即使是单核的也会在同一时间处理多个进程

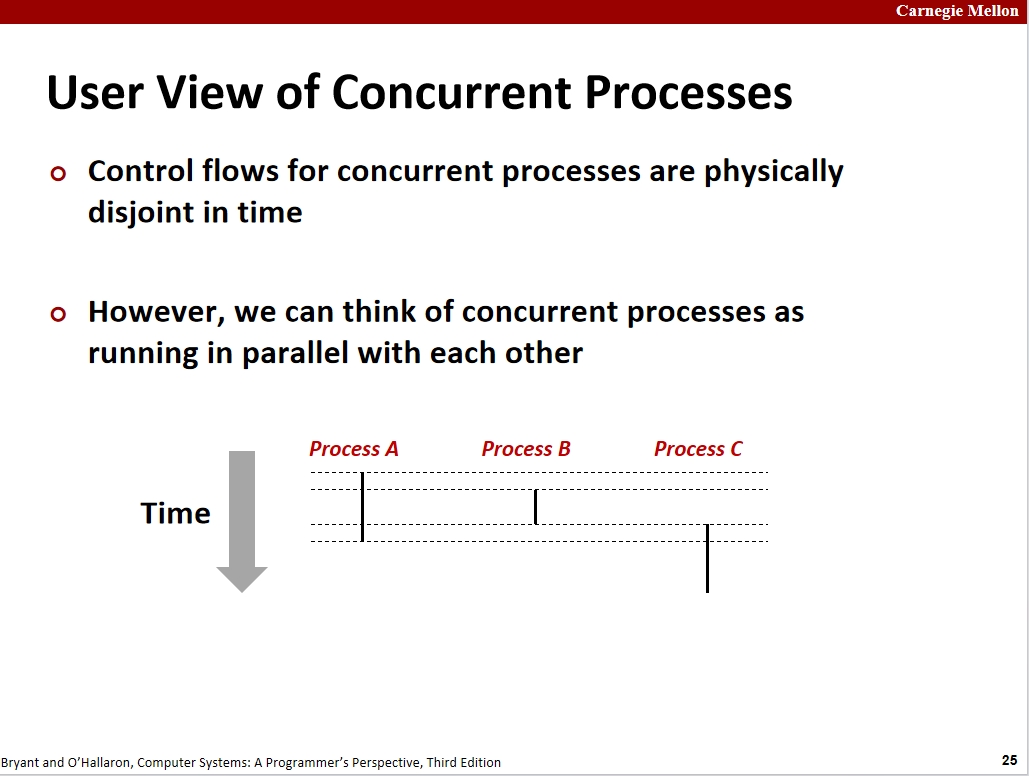


事实上，对于一个核心，该核心同一时间确实只能执行一个进程。但是，需要注意，我们前面提到了，计时器导致的异步中断，这里的中断可以在这里体现，在一个进程运行一段时间后，系统会将当前的进程的寄存器中的值保存到当前进程所对应的对应储存寄存器的地址中去，然后，系统会让寄存器切换当前要执行的进程，这个过程会进行进程中储存其中的寄存器中的值到实际寄存器中的值的操作等。由此可见，一个核心事实上确实一个时间只能执行一个进程，当时由于这种不断切换的特性，使得其能够给一种运行了多个线程的错觉，这其实是有计算机的高效性决定的，在现代的多核处理中也是如此，每个核心也还是一个时间处理一个任务，但由于数量多了所以这个切换算法会更加复杂

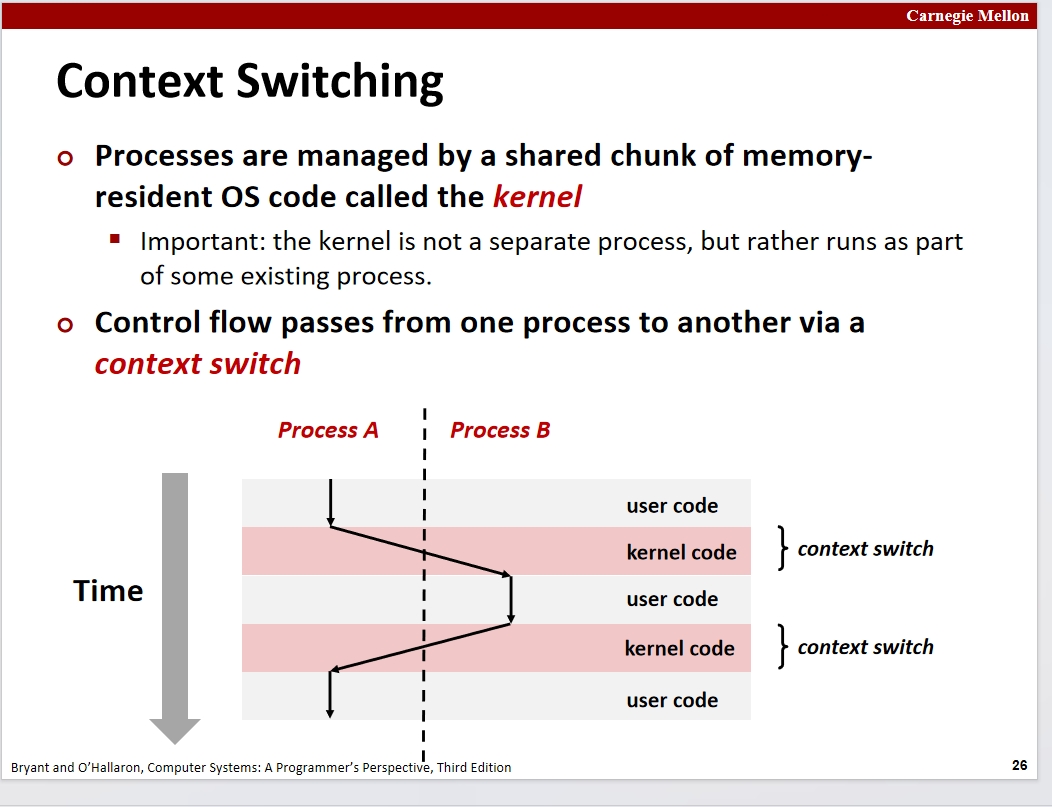
接下来再引入一个概念[逻辑控制流]，这个包含了这个进程的所有指令。还有，对于每个进程，我们应该知道它的rsp寄存器对应的值就是PC的值，也就是说，通过这个对PC的值的保存，我们能够实现进程切换时对程序运行状态的保存

当逻辑控制流在时序上是重叠时，称之为[**并发**]，否则称之为连续

对于这个的理解，在单核系统中，这个时序的重叠为在短时间内连续切换的多个进程，这些进程被视为并发的。而对于多核系统则更严格，这个只有真的在同一时间执行的多个进程，这几个进程才被视为并发的



对于这个**同一时间**需要进行一些分析，这个同一时间可以是不同时间，它的逻辑上应该是说在一段时间内如果一个进程形成的开始结束时间段与另一个进程形成的开始时间结束时间段间存在重合，就视为这俩个进程是并发的，以单核系统来看的话，只需要判断俩个进行的开始结束时间间是否存在重叠，其实就能够判断是否存在并发，也就是说，对于所有的系统应用，这些是与所有的进程都会并发执行的



接下来就可以分析下系统调用了，这里的话主要涉及到了内核，当一个进程的时间片用尽或者发生了计时器的触发等，内核会重新取得系统的控制权，接下来，内核会通过一系列算法决定接下来是谁来进行下一段时间片的使用，接下来，内核会将将要执行的时间片的对应数据给复制到核心中的寄存器及一些需要的位置中去。再之后，核心会将控制权再次交给接下来的进程，并等待再次取得控制权。

对于这些系统调用，在很多语言中其实都有一系列的对于这些系统级函数的抽象好的函数，对于这些函数，我们使用应该对其进行返回值检测，不然这个可能会出现一些我们不想要的错误

**进程的状态分析**

1~Running

该进程正在被运行并且执行内容或者这个进程可以被调度，也就是说这个进程可以处于现在不在执行的状态但需要处于一个可以被执行的状态，这是因为进程切换的操作

2~Stopped

该进程现在没在被执行，这意味着执行被暂停。通常一个进程收到一个信号后，这个进程以后不会被执行，转变为停止状态。、

这种一般有多种情况需要停止:  
 1~接收到一个默认是停止当前进程的信号

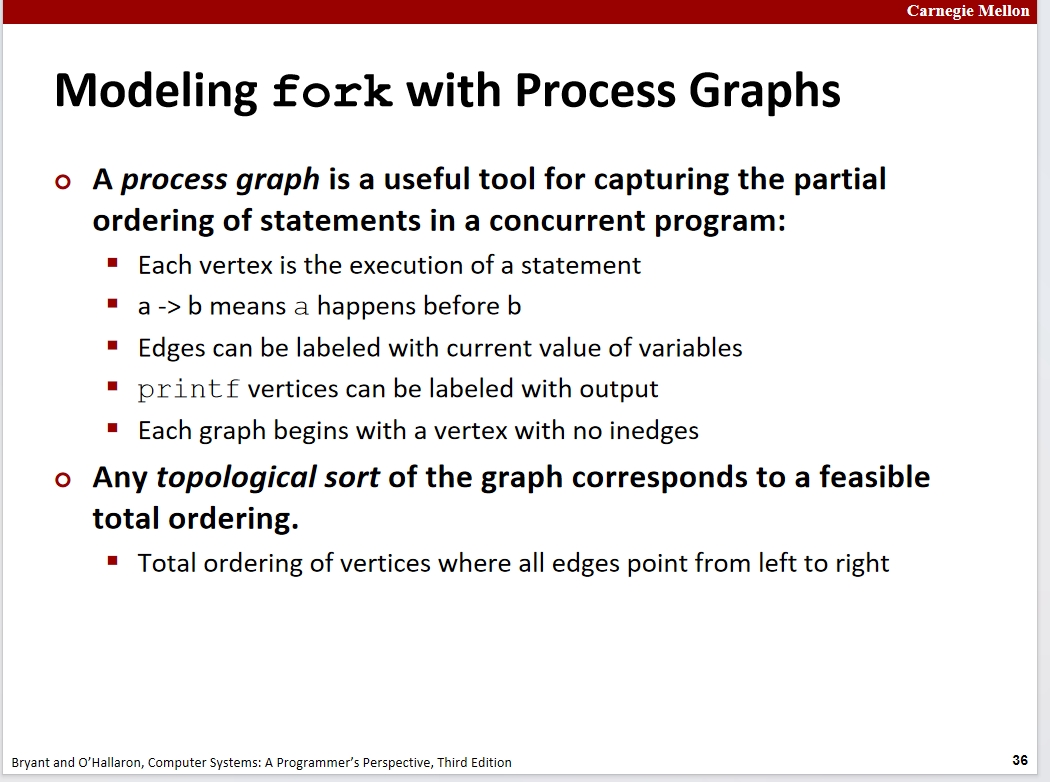
2~程序从主程序返回时，就比如当从一个main函数返回时， 系统会关闭当前进程，或者也可以通过exit函数关闭进程

进程创建(**fork**)

在程序中，我们可以使用fork函数来创建一个子进程，这个进程包含了父进程的所有信息且信息在创建时是相同的。对于子进程和父进程，当其结束时都会返回一个值。如果进程返回值为0，那么这个函数是在子进程中执行的，如果返回值是一段pid值，那么一般是在父进程中进行的。至于进程的先后顺序，我们无法确定到底是先运行哪个，这个是由操作系统在进程调度这方面决定的。

对于父子进程，他们使用相同的共享文件，如果再父子进程中都进行打印操作，可以看到在一个终端上会输出对应的父子进程的信息，我们可以通过这种共享部分操作来实现一些操作

当进程的创建逐渐增多时，这个的逻辑关系是相当复杂的，故这里引入一个**进程图**的工具进行处理分析。这个捕捉调用fork时可能发生的操作

关于进程图的一些定义属性如上图。

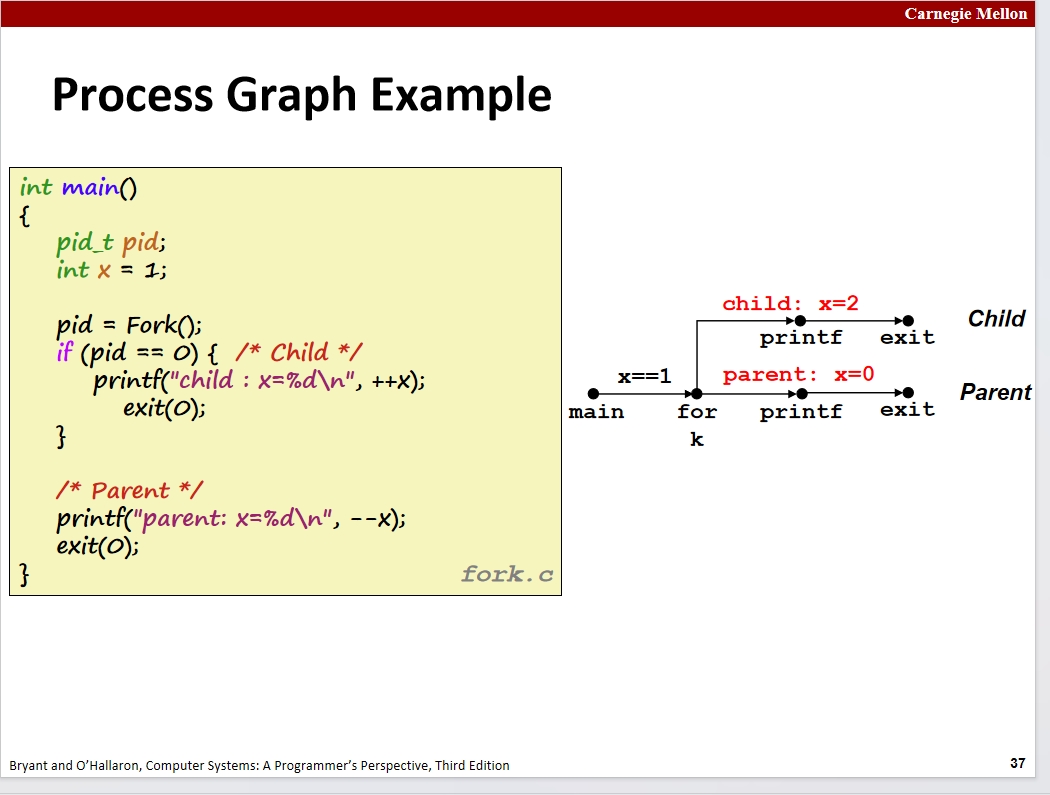
1~对于图中的每一个分支，都是一个进程

2~边在点之前发生。a->b表示a在b之前发生

3~使用当前变量标记边

4~诸如printf函数，这些函数可以用作图的节点 来代表

5~每个图形都从一个没有入边的顶点开始



如上右图中的进程图，在fork之后，其包含parent和child的俩个进程，接下来，代码中包含了一些不同进程中才会使用的函数，对于这些，一般可以用来作为进程的标识，通过对特定元素的观察及在进程图中的标志来进行区分。还有就是，每个进程图中的节点应该是一个具体的操作，就比如函数等

对于这些进程图中的执行顺序，每个之间包含着一些进程方面的调用和切换。这里可以使用拓朴排序来提供一个可选的执行顺序

需要注意下fork出来的子进程的执行代码，前面已知，fork出来的进程是会继承一系列父进程的东西的，其中也包括了程序计数器，也就就是说，fork出来的进程是不会再重头开始的，想想也应该这一点，不然就一直死循环且还浪费空间出不去了。

接下来分析下任何一个进程因为任何原因终止时可能出现的几种情况。

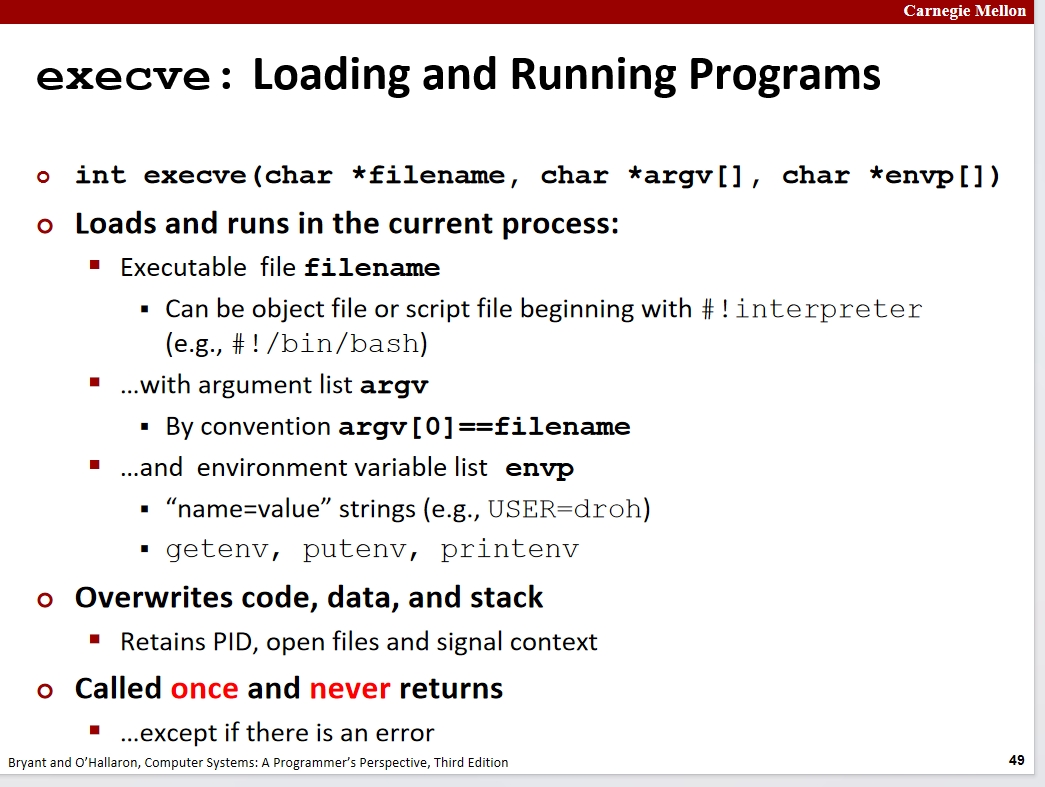
当进程终止时，系统会一直维持这个进程一直到其被

[回收raped],这个可能是因为这个进程结束时的一些状态信息 是父进程所想要的。就比如对于返回状态的检测，如果束后 直接就被回收，那么它其实是没有任何信息返回的。总的来说， 当一个进程结束时，它其实还没有完全的消失，这种状态被视 为一种[僵尸进程]。

对于这种[zombie]，系统会一直等待它被回收，这一般是通过一些函数实现的，就比如wait或者waitpid函数。此时父进程就被传递了子进程的返回状态，此后应该是通过一些系统调用来实现进程的回收的，通过内核来回收一个父进程的僵尸子进程。

但是，有时父进程并不会对其的子进程进行回收。但是程序应该是要对所有的进程进行回收的。这是就需要一些兜底的程序，这时系统会使用pid为1的进程来回收。这个pid为1的init进程总是会进行这样的对僵尸进程的回收。也就是说，这个有点像QT中的对象树机制。

如果一直没有对[僵尸进程]进程回收，那么他会一直驻留在内存空间里面，这一段内存将会是一直无法写入的，也就是说，这个无法回收的情况其实也是一种内存泄漏的情况。



接下来就进入到**execve**函数了

该函数可以在一个进程中中断程序，然后调用一个目标程序，参数类型在上图中，对于第一个参数，应该是目标文件的具体位置(相对路径或者绝对路径都可)，但是这个文件必须是一个可执行文件，你应该知道为什么。

第一个参数char\*filename通常会用到的是.shell文件，因为这个文件可以直接由终端读取执行，且在这个文件中，可以调用其他一些解释器来使用一些别的语言操作

首先先来看一下常用的.shell文件的格式

通常这类文件中的第一行以(**#!/**)起手，接着后面会跟着接下来整个文件中要使用的解释器的具体路径。这个解释器是用来解释你接下来这个文本文件中的代码的。

就比如如果使用#!/bin/bash，这个将会使用对应路径下的bash，这应该得是一个bash的解释器，接下来，这个解释器将会读取这个.shell文件中的接下来的代码并进行运行

第二个参数char \* argv()

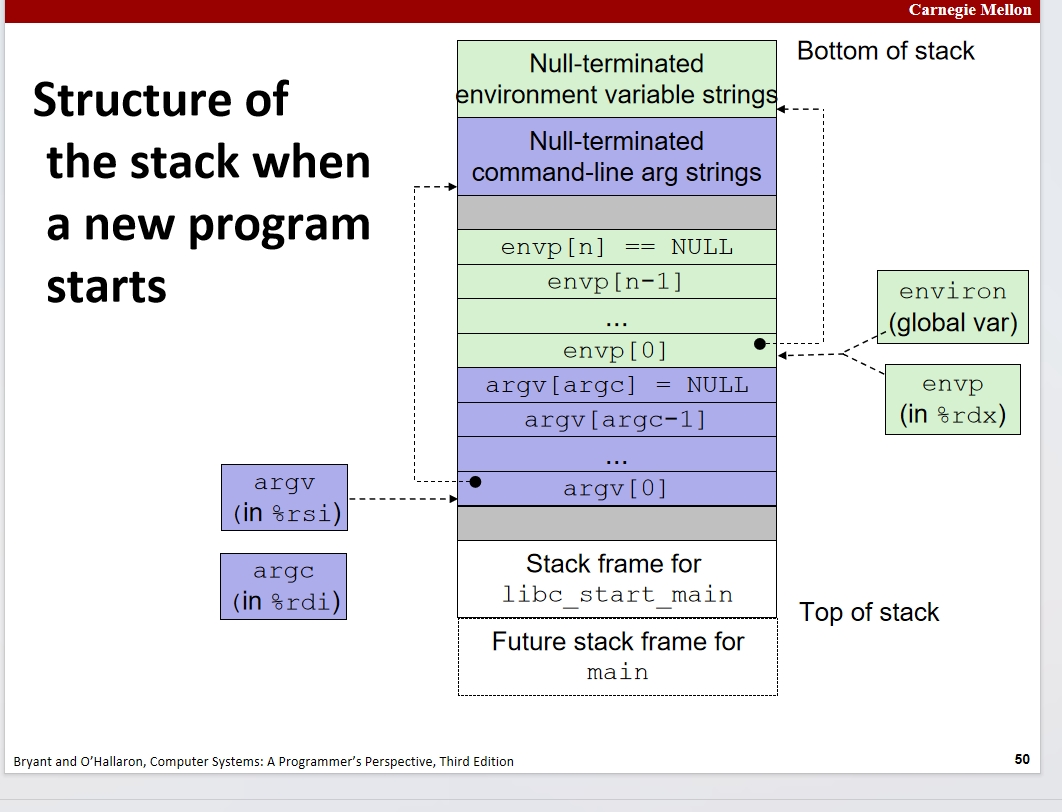
这个参数应该是正在执行的文件的名称，也就是当前执行的文件名，也就是说，对于一个程序，如果想要知道当前执行的文件名，通过打印argv[0]就可以实现

第三个参数char\*envp[]

这个参数是当前要执行文件的一系列的环境变量配置，当我们显示的进行传递时，它会将更新后的程序的环境变量设置为数组中的信息，不然，它会继承调用execve函数的主函数的环境变量

对于execve函数的调用，新调用的函数将会覆盖原有的所有数据，其保留的一般只有当前的进程id，也就是借皮上市。而且这个函数是永远不会返回的，也就是是说，一次调用，但永远不会返回时它的一个特性

接下来看具体的程序覆盖后的堆栈信息



首先，对于栈，main函数未来的栈帧将会位于这个栈顶，因此会位于最下，并位于libc\_start\_main之后，这是正确提到的启动程序，接下来堆栈还有一些填充，如上，对应的栈其实还有一些生长来储存对应的参数，其中也如之前所说，对应的argv参数会被储存在rsi中，但是只会储存这个数组的第一个元素，数组之后的元素还是被储存在堆栈中，argc参数数量也是如此，这个储存在对应的rdi寄存器中，其中，参数列表的终止位和环境列表的终止位都是一个NULL指针。

并且，我们程序的一个全局环境变量指针指向了对应的envp的起始位置，这个通常是第三个参数，因此会被储存在rdx寄存器中

从这里可以看出，通过对这些参数的检查，我们可以看到一些命令行调用的信息。而对于这些信息，我们可能需要进行接收等处理。但是我们一般不能在原进程中进行使用execve函数查看，因为这个会导致我们的程序被覆盖。我们的标准操作应该是，先fock一个子进程出来，同时传递所有的当前进程的参数信息进行execve函数调用，然后对这些参数进行使用，这样能够对我们的父进程进行一个保护。

这里应该这么理解，我们需要一个额外的功能，但是这个是当前进程无法实现，我们想要通过一个额外的文件来实现我们对应的功能。但是，我们应该保证当前进程的操作不被覆盖的话，最好就是先fork一个子进程，然后在这个子进程中调用execve函数来实现对应的功能，在实现功能的同时实现对父进程的保护。