**异常**

**异常：**控制流的突变，用来响应处理器状态中的某种变化（事件）。

**异常号：**系统中可能的每种类型的异常都分配了一个唯一的非负整数的异常号。其中一些号码是由处理器的设计者分配的，其他号码是由操作系统内核的设计者分配的。前者的示例包括被零除、缺页、内存访问违例、断点以及算术运算溢出。后者的示例包括系统调用和来自外部 I/0 设备的信号。

**异常表：**是一张跳转表，表目K包含异常K的处理程序代码的地址。

**异常表基址寄存器：**存储了异常表本身的起始地址。

异常与过程调用的不同：

1.返回地址可能不同。过程调用一定返回到下一条指令，但异常不一定。

2.触发异常时，会把一些额外的处理器状态压栈，以便于从异常处理程序返回时，恢复现场。（如果控制从用户程序转移到内核，则压到**内核栈**中而不是用户栈，因为恢复现场是在内核完成。）

3.异常处理程序运行在内核模式下。

异常分两种：异步异常是由处理器外部的 I/0 设备中的事件产生的，同步异常是执行一条指令的直接产物。

**异步：**中断

I/O设备通过向处理器芯片上的一个引脚发信号，并将异常号放到系统总线上，来触发中断，**这个异常号标识了引起中断的设备**。在当前指令完成执行之后，处理器注意到*中断引脚*的电压变高了，就从系统总线读取异常号，然后调用适当的中断处理程序。当处理程序返回时，它就将控制返回给**下一条指令**，就好像没有发生过中断一样。

**同步：**

**（1）陷阱（系统调用）**

陷阱是有意的异常，是执行一条指令的结果。就像中断处理程序一样，陷阱处理程序将控制返回到**下一条指令**。陷阱最重要的用途是在用户程序和内核之间提供一个像过程一样的接口，叫做系统调用。

**汇编指令syscall n 请求服务n。（n是系统调用号，不等于异常号）**

从程序员的角度来看，系统调用和普通的函数调用是一样的。然而，它们的实现非常不同。普通的函数运行在用户模式中，用户模式限制了函数可以执行的指令的类型，而且它们只能访问与调用函数相同的栈。系统调用运行在内核模式中，内核模式允许系统调用执行特权指令，并访问定义在内核中的栈 。

所有到 Linux 系统调用的参数都是通过通用寄存器而不是栈传递的。

按照惯例，**寄存器 %rax 包含系统调用号（之所以用%rax传参，是因为系统调用号比较特殊，直接决定跳转到哪种系统调用）**，%rdi、%rsi等寄存器包含最多6 个参数。

从系统调用返回时，寄存器 % rcx 和 % rll 都会被破坏， % rax 包含返回值，负数返回值表明发生了错误。

**（2）故障**

故障由错误情况引起，它可能能够被故障处理程序修正。当故障发生时，处理器将控制转移给故障处理程序。如果处理程序能够修正这个错误情况，它就将控制**返回到引起故障的指令**，从而重新执行它（例如缺页异常）。否则，处理程序返回到内核中的 abort 例程，abort 例程会**终止引起故障的应用程序**。

**（3）终止**

终止是不可恢复的致命错误造成的结果，通常是一些硬件错误。处理程序将控制返回给一个 abort 例程，该例程会**终止这个应用程序**。

进程

进程的经典定义就是一个执行中程序的实例。系统中的每个程序都运行在某个进程的上下文 (context) 中。

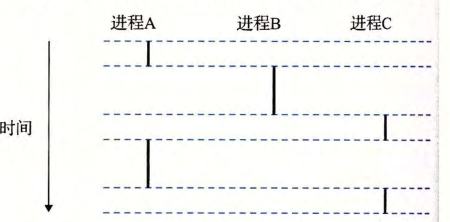
每次用户通过向shell 输入一个可执行目标文件的名字，运行程序时，shell 就会创建一个新的进程，然后在这个新进程的上下文中运行这个可执行目标文件。应用程序也能够创建新进程，并且在这个新进程的上下文中运行它们自己的代码或其他应用程序。

进程提供给应用程序的关键抽象：

• 一个独立的逻辑控制流，它提供一个假象，好像我们的程序独占地使用处理器。

• 一个私有的地址空间，它提供一个假象，好像我们的程序独占地使用内存系统。

**逻辑控制流**



**并发流：**从A开始到结束的时间里，若执行了B，则A、B是并发的（见上图的A与B）。单核也可以有并发。

**并行：**并发的真子集。如果两个流并发地运行在*不同的处理器核或者计算机*上，那么我们称它们为并行流。

**用户模式和内核模式：**处理器的控制寄存器中有一个模式位。

(1)若设置了模式位，进程就运行在内核模式（超级用户模式），例如异常处理程序。

(2)若没有设置模式位，则运行在用户模式中。用户程序必须通过系统调用接口间接地访问内核代码和数据。

(3) Linux 提供了一种聪明的机制，叫做 /proc 文件系统，它允许用户模式进程访问内核数据结构的内容.

**上下文：**上下文就是内核重新启动一个被抢占的进程所需的状态。

它由一些对象的值组成，这些对象包括通用目的寄存器、浮点寄存器、程

序计数器、用户栈、状态寄存器、内核栈和各种内核数据结构，比如描述地址空间的页表、包含有关当前进程信息的进程表，以及包含进程已打开文件的信息的文件表。

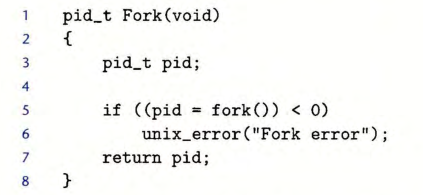
**上下文切换:** 1) 保存当前进程的上下文，2) 恢复某个先前被抢占的进程被保存的上下文，3) 将控制传递给这个新恢复的进程。

**错误处理包装函数：**给要调用的系统级函数做一个包装，接收其返回值并且对可能产生的错误进行处理。

例如给定如下包装函数，我们对 fork 的调用就缩减为1行。

不需要每次调用都要判断是否出错了。

pid = Fork() ;

****

**进程控制：**

**（1）PID**

每个进程都有一个唯一的正数ID(PID) 。规定PID非0是为了区分子进程。

getpid函数返回调用者的 PID 。

getppid 函数返回调用者的父进程的 PID。

**（2）创建进程**

*进制的状态：运行、停止（挂起）、终止。*

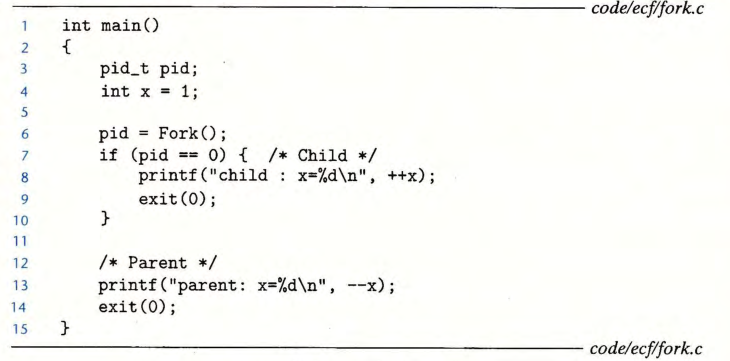
父进程通过调用fork 函数创建一个新的运行的子进程。

新创建的子进程几乎与父进程相同。子进程得到与父进程用户级虚拟地址空间相同的（但是独立的）一份副本，包括代码和数据段、堆、共享库以及用户栈。子进程还获得与父进程任何打开文件描述符相同的副本，这就意味着当父进程调用fork 时，子进程可以读写父进程中打开的任何文件。

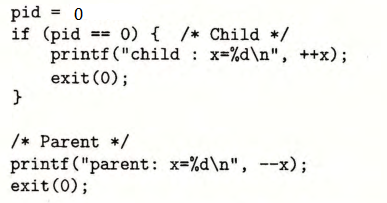
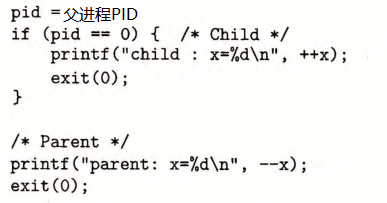
**父进程和新创建的子进程之间最大的区别在于它们有不同的 PID 。**

**在父进程中，fork 返回子进程的 PID。在子进程中，fork 返回 0 。**

因为子进程的 PID 总是非零，返回值提供了一个方法来分辨程序是在父进程还是子进程中执行。

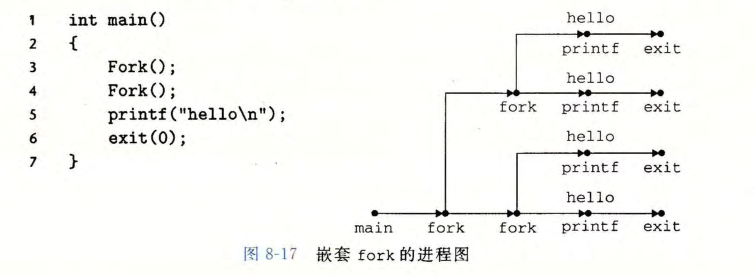


调用Fork后，下面两部分并发的执行。



**1.Parent和Child是并发执行的。**可能先执行一会Parent，再执行一会Child，也可能先执行一会Child，再执行一会Parent。

画进程图有助于理解带有嵌套 fork 调用的程序。



**2.地址空间相同但是独立。**例如，Parent对x的改动不会反映到child的x。

**3.共享文件。**运行这个示例程序时，我们注意到父进程和子进程都把它们的输出显示在屏幕上。原因是子进程继承了父进程所有的打开文件。当父进程调用fork 时，stdout 文件是打开的，并指向屏幕。子进程继承了这个文件，因此它的输出也是指向屏幕的。

**（3）回收子进程**

当一个进程由于某种原因终止时，内核并不是立即把它从系统中清除。相反，进程被保持在一种已终止的状态中，直到被它的父进程回收。当父进程回收己终止的子进程时，内核将子进程的**退出状态**传递给父进程，然后抛弃己终止的进程，

一个终止了但还未被回收的进程称为僵死进程 (zombie) 。

init 进程的 PID 为1,是在系统启动时由内核创建的，它不会终止，是所有进程的祖先。如果一个父进程终止了，内核会安排 init 进程成为它的孤儿进程的养父。

如果父进程没有回收它的僵死子进程就终止了，那么内核会安排 init 进程去回收它们。

一个进程可以通过调用 waitpid 函数来等待它的子进程终止或者停止。



1.参数pid：等待集合的成员

如果 pid>0, 那么等待集合就是一个单独的子进程，它的进程 ID 等于 pid 。

如果 pid= -1, 那么等待集合就是该进程所有的子进程。

2.参数options：改变默认行为。

options=0（默认）：挂起该进程，直到它的等待集合中的**一个**子进程终止。如果某个进程在调用时就已经终止了，那么 waitpid 就立即返回。

**返回值：已终止子进程的 PID。**

options= WNOHANG：如果等待集合中的所有子进程都没终止，那么立即返回0.

options= WUNTRACED：挂起调用进程的执行，直到等待集合中的一个进程变成巳终止或者**被停止**。

options=WCONTINUED: 挂起调用进程的执行，直到等待集合中一个正在运行的进程终止或一个被停止的进程收到 SIGCONT 信号重新开始执行。

可以用或运算把这些选项组合起来 。

3.参数statusp:

一个指针，waitpid 会在\*statusp中放上导致返回的子进程的**状态信息**

4. 错误条件

如果调用waitpid的进程没有子进程，那么它返回-1, 并且设置 errno 为 ECHILD

如果waitpid函数被一个信号中断，那么它返回-1，并且设置 errno 为 EINTR

5.wait是waitpid的简单版本。

**调用 wait(&status) 等价于调用 waitpid(-1, &status, 0)**

。

**（4）让进程休眠**

sleep 函数将一个进程挂起一段指定的时间。如果请求的时间量已经到了，sleep 返回0,否则返回还剩下的要休眠的秒数。后一种情况是可能的，如果因为sleep 函数被一个信号中断而过早地返回。

pause 函数让调用函数休眠，直到该进程收到一个信号。

**（5）加载并运行程序**

execve 函数在当前进程的上下文中加载并运行一个新程序。



execve 函数加载并运行可执行目标文件 filename,且带参数列表 argv 和环境变量列表 envp。

只有当出现错误时，例如找不到filename，execve 才会返回到调用程序，否则从不返回。

新程序的主函数有如下原型：

int main(int argc, char \*\*argv, char \*\*envp);

argc：argv[]数组的大小。

argv：指向argv[]（命令行参数 字符串数组）中第一个条目。

envp：指向envp[]（环境变量 字符串数组）的第一个条目。