第 2 章 进程与内核

2.1 简 介

操作系统的主要功能是给用户程序(应用程序)提供执行环境，这包括定义程序执行的基本框架，提供一组服务，例如文件管理和 I/O 操作，以及访问这些服务的接口。UNIX 操作系统具有一个能够有效地支持多种应用程序的丰富多样的编程接口[Kern 84]。本章介绍 UNIX 系统的主要组成部分以及它们之间如何通过彼此交互提供一个强大的编程环境。

UNIX 有许多不同的变体。其中主要有 AT&T 的 System V(System V 的最新版本 SVR4，现属于 Novell)，加州伯克利分校的 BSD，开放软件基金会的 OSF/1，及 Sun Microsystems 的 SunOS 和 Solaris。本章介绍传统 UNIX 系统(即基于 SVR2[Bach 86]，SVR3[AT&T 87]，4.3BSD[Leff 89]或更早版本)的内核和进程框架。UNIX 的变体 SVR4，OSF/1，4.4BSD 和 Solaris 2.x 和这个基本模型有很大不同；后续章节将对当前流行的版本进行详细介绍。

UNIX 应用开发环境包括一个基本抽象概念——进程。传统 UNIX 系统中，进程在一个地址空间上执行单一指令序列。进程地址空间包括可以访问或引用的内存单元的集合，进程控制点通过一个一般称为程序计数器(Program counter，PC)的硬件寄存器控制和跟踪进程指令序列。许多较新的 UNIX 版本支持多个控制点(称为线程，threads[IEEE 94])。因此在一个进程内可以有多个指令序列。

UNIX 系统是一个多道程序环境，即几个进程可以同时在系统中并发活动。系统为这些进程提供虚拟机的某些功能。在一个纯虚拟机结构中，操作系统给每个进程一个它是系统唯一进程的假象。程序员在写应用程序时，可以认为系统中只有他的代码在运行。在 UNIX 系统中，每个进程有自己的寄存器和内存，但必须通过操作系统才能进行 I/O 和设备控制。

进程地址空间是虚拟的①。通常只有部分映射到物理内存单元上。内核将进程地址空间的内容保存在各种存储对象上，包括物理内存，磁盘上的文件，特别地还可以保存在本地和远程磁盘的交换区上。通常由内核的内存管理子系统完成进程存储页面在这些对象之间的转移。

每个进程还有一组对应于实际硬件寄存器的寄存器。系统中有许多活动进程，但只有一组硬件寄存器。内核将当前运行进程的寄存器组保存在硬件寄存器中，将其他进程的寄存器组保存在每个进程的数据结构中。

进程间会竞争系统中的各种资源，例如处理器(也称为中央处理器或 CPU)，内存和外围设备。操作系统此时就担当起资源管理者的角色，优化的分配系统资源。进程不能立刻获得资源时，系统将其阻塞(暂停执行)，直到资源可用后再恢复执行。(CPU 是这样一种资源，在单处理器系统中，任一时刻只有一个进程可以运行在其上。其他进程都被阻塞住，等待 CPU 或其他资源。内核限制每个进程在 CPU 上运行一个小段时间(称为定额，一般为 10 毫秒左右)。然后再切换到另一个进程。通过这种方法，内核给每个进程一种并发运行的假象。这样，每个进程获得一段 CPU 时间并继续执行。这种操作方法称为时间片。

从另一个角度来看，计算机向用户提供各种设备，如处理器，磁盘，终端和打印机。应用程序员并不关心这些部件功能和结构的底层细节。操作系统拥有对这些设备的完全控制，并提供一个高级抽象编程接口，应用程序可以通过这些接口访问这些部件。这样就隐藏了硬件的所有细节，大大简化了程序员的工作②。 通过对这些设备的集中控制，还可以提供一些诸如访问同步(若两个用户在同一时刻想访问同一设备)和错误恢复等其他功能。应用编程接口 (application programming interface API)定义了用户程序用操作系统之间的所有交互语义。

我们已经对始将操作系统当作一个可以完成某些工作的实体。这个实体究竟是什么？一方面，操作系统是个程序(一般称为内核)，它可以控制硬件，创建，终止并控制所有进程(如图2-1)。更广义地讲，操作系统不仅包括内核，而且还包括一系列的其他程序和工具(例如 shell，编辑器，编译器和诸如 date，ls，who 之类的程序)，它们在一起组成一个非常有效的工作环境。显然.内核本身功用有限，而用户购买 UNIX 系统时希望得到许多类似于上面的那些程序。然而，内核在许多方面都非常特殊，它定义了对系统的编程接口，它是唯一不可缺少的程序，没有它什么都不能运行。虽然若干 shell 或编辑器可以并发运行，但内核一次只能加载一个。本书致力于研究内核，除非明确说明，当提到操作系统或 UNIX 时，一般都是指内核。

图 2-1 内核号进程及设备间的交互

将前面的问题换个问法，究竟什么是内核，它是一个进程吗？还是与其他所有进程都有所不同？内核是一个直接运行在硬件上的特殊程序，它实现了进程模型和其他系统眼务。它在磁盘的文件一般是/vmunix 或/unix(这取决于 UNIX 厂商)。当系统启动时，通过一个称为引导程序(bootstrapping)的特殊过程从磁盘上加载内核。内核初始化系统，为运行进程设立环境。然后它创建一系列初始进程，这些进程接着又创建其他进程。一旦被加载后，内核将一直保留在内存中，直到系统关闭为止，它管理所有进程并为它们提供各种服务。

UNIX 操作系统借助以下十种方式提供功能：

·用户进程通过 UNIX API 的内核部分，系统调用接口，显式地从内核获得服务。内核以调用进程的身份执行这种请求。

·进程的某些不正常操作，诸如除数为 0，或用户堆栈溢出将引起硬件异常。异常需要内核干预，内核为进程处理这些异常。

·内核处理外围设备的中断。设备通过中断机制通知内核 I/O 完成和状态变化。内核将中断视为全局事件，与任何特定进程都不相关。

·像 swapper 和 pagedaemon 之类的一组特殊的系统进程执行系统级的任务。比如，控制活动进程的数目或维护空闲内存池。

下面将介绍这些不同的机制并定义进程的执行上下义。

2.2 模式，空间和上下文

为了运行 UNIX，计算机硬件至少提供 2 种不同的执行模式 具有较高特权的内核态和特权较低的用户态。正如你所想的那样，用户程序在用户态执行，内核功能在内核态执行。内核保护地址空间的某些部分，防止用户态进行访问。此外，某些特权机器指令，如操作存储管理寄存器的指令等，它们只能在内核态执行。

许多计算机具有两种以上的执行模式。例如，Intel 80x86 体系结构有四层执行环(rings of execution)，内特权最高。但 UNIX 只使用了其中的两环。具有不同执行模式的主要原因是为了保护。由于用户进程在较低的特权级上运行.它们将不能意外地或故意地破坏其他进程或内核。程序错误造成的破坏被局域化，一般不会影响系统中的其他活动或进程。

大多数 UNIX 实现都使用了虚存系统。在虚存系统中，程序使用的地址不直接对应物理存储器的存储单元。每个进程有自己的虚拟地址空间，对虚存地址的引用通过地址转换表转换到相应的物理单元上。许多系统用页表来实现这些映射表，每一个表项对应进程地址空间的一页(内存分配和保护的单元，大小固定)。计算机的存储管理单元(MMU)一般有一组寄存器来标识当前运行的进程(也称当前进程)的转换表。在当前进程将 CPU 放弃给另一个进程时(一次上下文切换)，内核通过指向新进程地址转换表的指针加载这些寄存器。MMU 寄存器是有特权的。只能在内核态才能访问。这就保证了一个进程只能访问自己用户空间内的地址，而不会访问和修改其他进程的空间。

每个进程的虚拟地址空间中固定的部分是内核的正文和数据结构。这部分称为系统空间(system space)，或内核空间(kernel space)，它们只能在内核态访问。系统中只有一个内核实例运行，因此所有进程都映射到单一内核地址空间。内核中维护全局数据结构和每个进程的一些对象信息。后者包括的信息使内核可以访问任何进程的地址空间。由于 MMU 寄存器中有必要的信息，内核可以直接访问当前进程的地址空间。偶而地内核也须访问非当前进程的地址空间。而这是无法直接完成的，需通过特殊的临时映射实现。

尽管所有进程都共享内核，但其系统空间是受保护的，进程在用户态是不能访问的。进程不能直接访问内核，而必须通过系统调用接口。当进程调用一个系统调用(system call)时，其执习了了一个特殊的指令序列，使系统进人内核态(被称为模式转换)，并将控制权交给内核。由内核替代进程完成操作。当系统调用完成后，内核执行另一组特征指令将系统返回到用户态(另一个模式转换)，控制权返回给进程。系统调用接口在 2.4.1 节中深入讨论。

每个进程有两个重要的对象，虽然它们由内核管理，但一般在进程地址空间内实现。它们就是 u 区(也称用户区)和内核堆栈(kernel stack)。U 区是一个包含着有关进程的内核感兴趣的数据结构，如进程打开的文件表，标识信息及在进程不运行时所保存的进程寄存器值。进程是不允许随意更改这些信息的，因此 u 区在用户态是不可访问的(某些实现允许进程只读访问，不能更改 u 区)。

UNIX 内核是可重入的(re-entrant)，这就是说多个进程可并发参与内核活动。实际上，它们甚至可以并行地执行同一个子程序。当然，实际上某一时刻只有一个进程执行，其他的被阻塞或等待运行。)因此每个进程需要有自已的私有内核栈来记录它在内核中调用的函数的序列。许多 UNIX 的实现都是在每个进程的地址空间内分配内核堆栈，但不允许从用户态访问它。尽管 u 区和内核堆栈是属于每个进程空间的实体，但从概念上讲它们是属于内核的。

另一个重要概念是执行上下文(execution context)。内核函数既可以在进程上下文运行，又可以在系统上下文运行。在进程上下文中，内核代表当前进程执行(例如，执行一个系统调用)，可以访问和修改进程的地址空间，u 区及其内核堆栈。此外，若进程必须等待资源或设备活动，内核可以阻塞当前进程。

内核也必须完成系统级任务，如响应外设中断和重新计算进程优先级。这些任务并不是为了某个特定进程完成的，因此在系统上下文中处理(也称为中断上下文)。当运行于系统上下文时，内核可能不会访问当前进程的地址空间，u 区或内核堆栈。在系统上下文中执行时，内核不会阻塞，否则就会阻塞一个无辜的进程。在某些情况下，系统中可能没有一个当前进程，如所有进程因等待系统 I/O 完成而阻塞。

现在，我们已经说明了用户态与内核态，进程空间与系统空间以及进程上下文和系统上下文之间的区别。图 2-2 总结了上述概念。用户代码在用户态和进程上下文中运行，并只能访同进程空间。系统调用和异常在内核态而不是在进程上下文中处理，并可访问进程和系统空间。中断处理在内核态和系统上下文运行，只能访问系统空间。

2.3 进程抽象

究竟什么是 UNIX 进程？一个经常被引用的答案是“进程是程序运行的一个实例”。透过这种浮于表面的定义，研究一下进程的各种特征对我们会更有帮助、进程是一个运行程序并为其提供执行环境的实体。它包括一个地址空间和一个控制点。进程是基本调度实体—一任何时刻只有一个进程在 CPU 上运行。同时，进程竟争并占用各种系统资源，如设备和内存。它还向系统请求服务，由内核来为其完成。

进程有确定的生命周期 － 大多数进程通过 fork 或 vfork 系统调用创建，并一直运行直到调用 exit 后终止。在它的整个生命期内，进程可能运行一个或多个程序(一般是一次一个程序)。通过系统调用 exec 运行一个新的程序。

图 2-2 执行模式及上下文

UNIX 进程有一个严格定义的层次结构。每个进程有一个父进程，并可以有一个或多个子进程。进程的层次结构很像一棵倒立的树，其顶端为 init 进程。init 进程(由于它执行程序/etc/init 才这样命名)是系统启动后创建的第一个用户进程。它是其他所有进程的祖先。某些系统进程，如 swapper 和 pagedaemon(也称为 pageout daemon)，是在系统启动过程中创建的，它们不是 init 进程的后代。当某些有活动子进程的进程终止时，其所有活动的子进程都变成孤儿(orphans)，并成为 init 进程的子进程。

2.3.1 进程状态

任何时刻，UNIX 进程总处于某一严格定义的状态。它们响应各种事件，从一个状态转移到另一个状态。图 2－3 描述了 UNIX 中重要的进程状态和引起状态转移的事件。

系统调用 fork 创建一个新的进程，它一开始是初始状态(也称为空闲状态)。当进程完全创建后，fork 把它转变为就绪(ready to run)状态，它必须等待被调度。最终内核将挑选其运行，也就是要发生一次上下文切换。系统调用内核例程(一般称为 swtch())，它将进程的硬件上下文(参考 2.3.2 节)加载到系统寄存器中，并将控制权转交给那个进程。从这时开始，新的进程的行为将和其他进程一样，进行如下所述的状态转移。

运行于用户态的进程同系统调用或中断进人内核态，并在它们完成后返回用户态③ 当执行系统调用时，进程可能会等待一个事件或一个当前不可用的资源。它通过调用 sleep()实现等待，sleep()将进程加入睡眠进程队列中，并把其状态改为 asleep。当事件发生或资源可用时，内核唤醒进程，现在进程处于就绪态，并等待系统调度其运行。

当进程被调度运行时，它开始运行于内核态(内核运行态)，在那里完成上下文切换。而其以后的转移过程是由其被切换前的行为所决定的。如果进程是新创建的或正在执行用户代码(因更高优先级的进程运行而被换下几它立即返回用户态。若它是因在执行系统调用时等待资源而阻塞，它会恢复执行内核态的系统调用。

图 2-3 进程状态及其转换图

最终，进程通过调用 exit 系统调用，或因一个信号(信号是由内核发出的通知——见第 4 章)而终止。在这两种情况中，内核会释放除了退出状态和资源使用信息外的所有进程资源，并将进程变为僵尸状态。进程在其父进程调用 wait(或其变体)之前会一直保留这些状态，wait 会删除进程并向父进程返回退出状态(见 2.8.6 节)。

4BSD 添加了一些 SVR2 和 SVR3 中并不支持的状态。进程可因暂停信号(SIGSTOP，SIGTSTP,SLGTTIN 或 SIGTTOU)进入暂停(stopped)或挂起(suspended)状态。其他信号仅当程序在运行时才会被处理。与此不同,暂停信号立即改变进程的状态。若程序处于运行或就绪态，状态变为暂停态。若进程处于睡眠时产生了这个信号，它的状态变成睡眠和终止。一个暂停进程可以由继续信号(SIGCONT)重新启动，并返回就绪态。若进程在睡眠态是被暂停，SIGCONT 将进程状态转移到睡眠态。System V UNIX 在 SVR4(见 4.5 节)中加入了这些特性④。

2.3.2 进程上下文

每个进程都有一个严格定义的上下文，包括描述这个进程的所有信息。上下文由几个部分组成：

· 用户地址空间 一般划分为几个部分——程序正文(可执行代码)，数据，用户堆栈，共享内存区，等等。

·控制信息 内核使用两个数据结构维护进程的控制信息——u 区和 proc 结构。每个进程也有它自己的内核堆栈和地址转换表。

·凭证 进程的凭证包括与其相关的用户 ID 和组 ID，这在 2.3.3 节中进一步讨论。

·环境变量 这是一些形如：variable=value 的字符串组，是从其父进程继承过来的。大多数 UNIX 将其存在用户栈底。标准输入输出库提供对这些变量的增加，删除，更改以及将从变量中取值的函数、当调用一个新程序时，调用者用 exec 取得原始环境或提供一组新的变量。

·硬件上下文 这包括通用寄存器中的值以及一组特殊的系统寄存器、系统寄存器包括：

·程序计数器(PC)，它记录将要执行的下一指令地址。

·堆栈指针(SP)，它包括栈顶元素的地址⑤。

处理器状态字(PSW)，它包括几个表明系统状态信息的状态位，如当前和以前的执行模式，当前和以前的中断优先级，溢出以及进位位。

·内存管理寄存器，它对应进程的地址转换表。

·浮点单元(FPU)寄存器。

程序寄存器包括当前正在运行进程的硬件上下文。当发生上下文切换时，寄件器中的值都保存到当前进程 u 区(称为进程控制块，PCB)的特定部分。内核选择一个新进程运行时，将从 PCB 中装载硬件上下文。

2.3.3 用户凭证

系统用一个唯一的号码标识每一个用户，这个号码称为用户 ID 或 UID，系统管理员也创建若下用户组，每组都有一个不同的用一组 ID 或 GID。这些标识符影响文件所有权和访问权限，以及向其他进程发送信号的能力。这些属性合在一起称为凭证。

系统识别一个称为超级用户的特权用户(通常就是 root 帐号登录的用户)。超级用户的 UID 为 0，GID 为 1。超级用户有许多一般用户所没有的特权。她或他可以访问其他入的文件，不论其是否有保护，而目可以执行一系列特权级系统调用(例如 mknode，它创建特殊的设备文件)。许多像 SVR4.1/ES 这件的现代 UNIX 系统支持增强型安全机制[Sale 92]。这些系统将下同的操作分给若干特权级，藉此来代替单一个超级用户。

每个进程有两对 ID ——真实的和有效的。当用户登录时，登录程序依据口令数据库(/etc/passwd 文件或某种分布机制，例如 Sun 公司的网络信息服务(NIS))中的 UID 和 GID 设置这一对值。当进程派生时，子进程从父进程中继承它的凭证。

有效 UID 和有效 GID 影响文件创建和访问：当文件创建时，内核将文件的所有权属性设置为创建进程的有效 UID 和 GID。在文件访问时，内核使用进程的有效 UID 和 GID 判别它是否有权访问文件(详见 8.2.2 小节)。真实 UID 和真实 GID 标识进程的真正所有者，影响其发送信号的权限，若进程没有超级用户特权，它只能向那些满足如下条件的进程发送信号，当且仅当发选者的真实 UID 或有效 UID 与接收者的真实 UID 匹配时。

有二个系统调用可以改变凭证。若一个进程调用 exec 运行一个 suid 模式的程序(参考 8.2.2 节)，内核将这个选程的有效 UID 改为那个文件所有者的 UID。同样，若程序是 sgid 模式的，内核将改变调用进程的有效 GID。

UNIX 提供这些特性是在一些特殊任务中赋于用户某些特殊权限。典型的例子就是 passwd 程序。它允许用户修改它自己的口令。这个程序必须对口令数据库实行写操作，而用户是不能对其进行直接修改的(以防他们改变其他用户的口令)。因此 passwd 程序属于超级用户，并设置了 SUID 位，这样就使用户在运行 passwd 程序时拥有了超级用户的特权。

用户也可通过调用 setuid 或 setgid 改变自己的凭证。超级用户可调用这些调用改变真实和有效 UID 或 GID。普通用户通过这些系统调用只能将他们的有效 UID 和 GID 改为真实的。

System V 和 BSD UNIX 中处理凭证有所不同。SVR3 还维护了保存 UID(saved UID)和保存

GID(saved GID)，它们是在调用 exec 之前的有效 UID 和 GID 值。setuid 和 setgid 调用可以将有效 ID 恢复为这些保存值。4.3BSD 不支持这一特性，但它允许用户属于一组附加组(使用 setgroup 系统调用)。尽管用户创建的文件属于他(她)的主组(primary group)，但用户不仅可以访问属于主组的文件，而且可以访问属于附加组的文件(假设文件允许组成员访问)。

SVR4 中同时加入了上述特性。它不仅支持附加组，而且在 exec 调用中实现了保存 UID 和 GID。

2.3.4 u 区和 proc 结构

每个进程的控制信息都有两个数据结构中——u 区和 proc 结构，在许多实现中，内核中有一个固定大小的 proc 结构数组，称为 proc 表。这个数组的大小严格的限制了任一时刻最多的进程数。较新的版本如 SVR4 允许动态分配 proc 结构，但有一个大小固定的指向它们的指针数组。由于 proc 结构在系统空间中，尽管有时进程不是运行的，任何时刻这些结构对内核中都是可见的。

u 区，或称用户区，是进程地址空间的一部分，即，仅在进程运行时它才被映射且可见。许多实现中，u 区总是被映射到每个进程虚拟空间的相同固定位置，内核只需简单的通过变量 u 进行访问。上下文切换的一个任务就是要重新建立这个映射，使内核中对 i 的引用指向新的 u 区的物理位置。

偶尔内核也要访问其他进程的 u 区。这是可能的，但只能间接地使用一组特殊映射完成。这些在访问语义上的不同决定了什么信息应该存放在 proc 结构中，什么应该存放在 u 区中。u 区中只包括那些在进程运行时才需要的数据。而 proc 结构中的信息可能在进程不运行时也需要。

u 区中的主要域包括：

- 进程控制块 ——保存进程不运行时硬件上下文。
- 指回这个进程 proc 结构的指针。
- 真实和有效 UID 和 GID⑥。
- 当前系统调用的参数，返回值或错误状态。
- 信号处理程序及相关信息(参考第 4 章)。
- 由程序头中获得的信息，如正文，数据，堆栈大小和其他内存管理信息。
- 打开文件描述符表(8.2.3 节)。现代 UNIX 系统，如 SVR4，可以按需动态地扩展这张表。
- 指向当前目录的 v 节点和控制终端的指针。v 节点代表文件系统对象，8.7 节中进一步讨论。
- CPU 使用统计，概况统计信息，磁盘限量和资源限制信息。
- 许多实现中，每个进程的内核堆栈是 u 区的一部分。

proc 结构中的主要域如下：

- 标识：每个进程有一个唯一的进程 ID(PID)并属于一个特定的进程组。较新的版本还分配给每个进程一个会话 ID。
- 当前进程中 u 区的内核地址映射表位置。
- 当前进程状态。
- 在调度队列，阻塞队列或睡眠队列中的向前指针和向后指针。
- 阻塞进程的睡眠通道(7.2.3 节)。
- 调度优先级和相关信息(第 5 章)。
- 信号处理信息：各种信号，包括忽略信号、阻塞信号、已发送信号及已处理信号的掩码(第 4 章)。
- 内存管理信息。
- 将这个结构链接到活动进程，空闲进程或僵尸进程队列的指针。

· 各种标志。
· 将结构链接到基于 PID 的哈希队列(hash queue)中的指针。
· 层次信息，描述进程与其他进程间的关系。

图 2-4 给出了 4.3BSD UNIX 中的进程关系。描述这种关系的域是 p_pid(进程 ID),p_ppid(父进程 ID)，p_pptr(指向父进程的 proc 结构)，p_cptr(指向最大的子进程)，p_ysptr(指向下一个更小的兄弟进程)和 p_osptr(指向下一个的更大兄弟进程)。

许多现代 UNIX 变体已经改变了这种进程抽象，在一个单一进程中支持几个线程。我们将在第 3 章中详细讨论。

2.4 内核态下运行

有三件事件会导致系统进人内核态——设备中断，异常，自陷或软中断。在每一种情况下，当内核接收到控制权，它依赖于派遣表控制转移方向。派遣表中包括处理这些事件的底层例程的地址。在调用相应的例程前，内核在内核栈上保存被中断进程的某种状态(如它的程序计数器，处理器状态字)。当例程完成后，内核恢复进程的状态，并将执行模式恢复为原来的模式(在内核态发生中断时，中断处理程序完成后仍返回内核态)。

这里有必要区分中断和异常。中断是由外围设备，如磁盘，终端或硬时钟引起的异步事件。由于中断不是当前正在运行的进程引起的，它们必须在系统上下文中被处理，而不可以

图 2-4　4.3BSD UNIX 中典型的进程层次

访问进程地址空间和 u 区。同理，它们也不能被阻塞，否则会阻塞其他进程。异常对进程而言是同步的，是由进程自身相关的事件引发的，如除 0 或访问非法地址。因此异常处理程序应在进程上下文中运行，它可访问进程的地址空间和 u 区，必要时可以被阻塞。软件中断或自陷，在系统执行特殊指令时出现，如有系统调用，在进程上下文中同步处理。

2.4.1 系统调用接口

系统调用的集合定义了内核向用户进程提供的编程接口。缺省链接到所有用户程序的标准 C 语言库为每个系统调用设置了一个包装例程。当用户程序调用一个系统调用时，就会调用相应的包装例程。这个例程将系统调用号(它标识内核的特定系统调用)压入用户栈，然后调用特殊的 trap 指令。这个指令的实际名字是与机器相关的(例如，MIPS R3000d 的 syscall，VAX-ll 的 chmk 或 Motorola 680x0 的 trap)。这条指令的功能是将执行模式变为内核态.并将控制权转移到定义派遣表中的系统调用处理程序。这个一般称为 syscall()的处理程序是内核中所有系统调用过程的起点。

系统调用在内核态中执行，但却是进程上下文。因此它可以访问进程地址空间和 u 区。由于它运行于内核态，它使用调用进程的内核栈。syscall()将系统调用参数从用户栈拷贝到 u 区，并在内核栈上保存硬件上下文。然后它使用系统调用号作为系统调用派遣向量(一般为 sysent[])的索引，确定内核函数应执行哪个特定的系统调用。当那个函数返回后，syscall()在对应的寄存器中设置返回值或错误状态，恢复硬件上下文，返回用户态，将控制权还给库例程。

2.4.2 中断处理

机器上中断的主要功能是允许外围设备与 CPU 交互，通知 CPU 任务完成，错误状态或其他急需注意的事件。这些中断的产生与正常系统活动是异步的(如系统不知道在指令流的哪一处会发生中断)，并且一般与任何特定进程无关。被调用处理中断的函数称为中断处理程序或中断服务例程。处理程序运行在内核态并处于系统上下文中。由于被中断的进程一般

与中断没有关系，处理程序应注意不要访问进程上下文。由于同样的原因，中断处理程序不允许被阻塞。

然而对中断进程还是有一些影响。尽管中断与进程无关，中断服务所使用的时间被计入进程的时间片中。同时，时钟中断处理程序计算当前进程的时标(两个时钟中断之间的时间)。因此需访问当前进程的 proc 结构。有一点非常值得注意，即进程上下文并未明确受到保护以防止中断处理程序对其进行访问。一个不正确的处理程序有可能破坏进程地址空间的任何部分。

内核也支持所谓的软中断或自陷，它们是由执行特殊的指令触发的。这些中断用来完成诸如触发上下文切换或调度低优先级的时间相关任务。尽管这些中断同一般系统活动同步，它们的处理同一般中断相似。

由于有不同的事件可能引发中断，所以一个中断可能在另一个中断被服务时发生。UNIX 系统对这种需求的方法是，将中断区分为不同的中断优先级，允许高优先级的中断可抢占低优先级的中断服务。例如硬件时钟中断必须高上网络中断，因为后者需要大量数据，并跨越几个时标。

UNIX 系统给每一种类型的中断一个中断优先级(interrupt priority level，ipl)。早期 UNIX 实现的 ipl 范围从 0～7。BSD 将其扩展为 0-31。处理器状态寄存器(processor status register)一般保存当前(或前面)⑦的位域。正常内核与用户进程运行于基本冲 ipl 上。中断优先级的号码不仅在不同的 UNIX 变体上有所不同，而且在不同的硬件体系结构上也不相同。某些系统中，ipl0 是最低优先级，而在其他系统中都是最高的。为了给内核和设备驱动程序开发者以便利，UNIX 系统提供了一组阻塞和非阻塞中断的宏。但不同的 UNIX 变体使用不同的宏达到相同的目的。表 2-1 中列举了一些在 4.3BSD 和 SVR4 中使用的宏。

表 2-1 4.3BSD 和 SVR4 中设置中断优先级的操作

当中断发生时，如果它的优先级高于当前的中断优先级，当前处理就被挂起，并调用新中断处理程序。中断处理程序在新的 ipl 上开始执行。当处理程序结束时，ipl 降低为上一个值(这通过保存在中断栈上的旧的处理器状态字取得)，内核恢复被中断的进程。内核收到中断的 ipl 低于或等于当前中断 ipl，这个中断并不立即处理，而是存储在中断寄存器中。当你降到足够低时，这个保存的中断再重新被处理。这一过程如图 25 所述。

图 2-5 中断处理

ipl 的比较及硬件设置都是以机器相关方式处理的。UNIX 也为内核显式地提供一种检查和设置 ipl 的机制。例如内核在处理某些临界区代码时，可以升高 ipl 以阻塞中断。这些将在 2.5.2 节中有深入讨论。

某些机器为所有中断处理程序提供了一个单独的全局中断栈(interrupt stack)。在没有中断栈的机器上，中断处理程序在当前进程的内核栈上运行。它们必须保证其余的内核栈同这个处理程序隔绝。内核是通过在调用处理程序之前在内核栈上压入一个上下文层(context layer)实现这一目的的。与栈结构相似，这个上下文层包含句柄返回时恢复先前执行环境的信息。

2.5 同步

UNIX 内核是可重入的。任何时刻都可能会有若干进程同时在内核中活动。但它们中只有一个(在单处理器中)是实际运行着的；其他进程被阻塞，等待 CPU 或其他资源。由于它们共享内核数据结构的同一副本，为了防止内核崩溃，有必要加上某种形式的同步。

图 2-6 所示是一个没有同步情况下会发生的例子。假设一个进程希望从链表中删除元素 B。它执行了第一行代码，但在执行下一行代码之前被中断，这时另一个进程得以执行。若第二个进程访问同一链表，它将发现该链表处于不一致状态，如图 2-6(b)所示。显然，我们应确保这些情况不会发生。

图 2-6 从链表中删除一个元素

UNIX 使用几个同步技术。第一条防线是 UNIX 内核是不可抢占的。这意味着任何一个在内核中执行的进程，尽管其时间片可能已用完，也不能被其他进程抢占。这个进程必须是自愿放弃 CPU。这种情况通常是进程在等待资源或事件被阻塞时，或是当它完成内核态活动后准备返回用户态时发生。上述两种情况下，因为 CPU 是自愿放弃的，进程可确保内核处于一个一致的状态。具有实时性的现代 UNIX 内核允许在一定条件的重入(在 5.6 节有详细讨论)。

内核的非抢占特性为大多数同步问题提供全面而彻底的解决方案.如图 2-6，当没心抢占问题时，内核管理链表时就无需加锁。但有三种情况同步仍是必需的-- 阻塞操作，中断和多处理器同步。

2.5.1 阻塞操作

阻塞操作会阻塞进程(在操作完成时使进程处于 asleep 状态)。由于内核是非抢占的，它可以无忧无虑地管理大多数对象(数据结构和资源)，没有其他进程会破坏它们。但还有一些对象，在阻塞操作过程中必须受到保护，这就需要额外的机制。例如，进程开始将数据从一个文件读取到内核内存区的磁盘缓冲区中。因为需要磁盘 I/O，进行必须等待 I/O 完成，同时允许其他进程运行。然而，由于缓冲区尚处于不一致状态，内核必须确保其他进程不能以任何方式访问这个缓冲区。

为保护这种对象，内核为其关联一把锁。锁可以是一个简单的位标志，加锁时对其进行置位，解锁时清位。任何进程在使用这个数据对象之前都必须检查其是否加锁。若已加锁，它必须等待该对象被解锁。若未加锁，它就对对象加锁并开始使用。一般而言，内核同时为对象关联一个需求(wanted)标志。当进程需要该数据对象但发现其已被加锁时，设置这个标志。当进程准备释放一个加锁对象时，它检查需求标志看是否其他进程等待使用这个对象，若有则唤醒这些进程。这个机制允许进程在很长一段时间内锁住资源，而在加锁期间该进程可以被阻塞并允许其他进程同时运行。

图 2-7 给出了对资源加锁的算法。以下几点需要注意：

· 当一个进程不能获得资源或等待诸如 I/O 完成这类事件时，进程阻塞自己。这是通过调用 sleep()完成的，这称为阻塞于资源或事件。

·sleep()将进程放在特殊的阻塞进程队列上，将其状态置为睡眠，并调用函数 swtch()开始上下文切换，允许其他进程执行。

· 进程释放资源后调用 wakeup()唤醒等待这个资源的所有进程⑧。wakeup()找到每一个这样的进程，将其状态变为可运行状态(runnable)，并把它加入调度队列等待系统调度。

· 一个进程从被唤醒到其被调度执行之间可能有一个很长的延时。其他进程可能会在这期间运行，并可能会再次对同一资源加锁。

· 因此在被唤醒运行后，进程仍需对资源进行检查，看其是否确实可用，若不可用则再次睡眠。

2.5.2 中断

尽管内核不会被其他进程抢占，但一个正操作内核数据结构的进程却可被设备中断。若这个中断处理程序正好也要访问那些数据结构，它们就可能会处于不一致状态。处理这个问

题的方法是在访问这些数据结构时屏蔽中断。内核使用诸如表 2-1 中的那些宏显式地提高 ipl 来屏蔽中断。这种代码段你为临界区(critical region)(见例 2-1)。

实例 2-1 在临界区内屏蔽中断

```
int x = splbio();                    /* rases ipl, returns previos ipl */
modify disk buffer cache;
splx(x);                             /* restores previous ipl */
```

图 2-7 资源加锁算法

中断屏蔽时要注意以下几点：

· 中断一般要求快速处理，不应受到严重干扰。因此，临界区应少而短。

· 只有那些可能访问临界区中数据的中断才必须屏蔽。在上面那个例子中，只有磁盘中断需被屏蔽。

· 两个不同的中断可以有相同的优先级。例如，许多系统中终端和磁盘中断的中断优先级都是 21。

· 屏蔽一个中断会将所有相同优先级或更低优先级的中断屏蔽。

注意：在描述 UNIX 系统时(block)这个词有许多不同用法。当一个进程系统等待资源可用或事件发生而进入睡眠称为进程阻塞于资源或事件。进程屏蔽(block)中断或信号使其暂时不能传递。最后，I/O 子系统与存储设备间按固定大小的块传输数据。

2.5.3 多处理器

由于内核所具有的最基本保护方式非抢占特性不复存在，多处理器系统有一类新的同步问题。在单处理器，由于不可抢占，内核可以毫无顾忌地处理大多数数据结构。它只需保护那些中断处理程序要访问的数据结构，或是调用 sleep()需要保持一致的数据。

在多处理器中，两个进程会在不同处理器上同时处于内核态，甚至并发地执行同一函数。因此，任一时刻当内核访问某一全程数据结构时，必须对其加锁以防其他处理器访问。这种加锁机制必须是多处理器安全的。若有不同处理器上运行的两个进程同时希望对某一对象加锁，只能有一个能成功地获取锁。

由于有多个处理器处理中断，对中断的保护更加复杂。通常阻塞每个处理器上的中断不是很好的方法，因为这可能会极大地降低性能。显然多处理器需要更复杂的同步机制。第 7 章详细分析这些问题。

2.6 进程调度

CPU 是一个由所有进程共享的资源。内核中在进程间分配 CPU 时间的那部分称为调度器(scheduler)。传统 UNIX 调度器使用抢占式轮转调度、相同优先级的进程以轮转方式调度，每个运行一个固定的时间片(通常是 100 毫秒)。若有一个更高优先级的进程准备就绪，无论当前进程是否用完其时间片，它都会被那个高优先级进程抢占(除非当前进程正在内核态运行)。

传统UNIX中系统,进程优先级由两个因素决定的—nice值和usage因子。用户通过nice系统调用更改进程的 nice 值，影响进程的优先级(只有超级用户可以增加进程优先级)。usage 因子是对进程近期使用 CPU 的度量。它允许内核动态的改变进程优先级。当进程不在运行时，内核定期增加它的优先级。当进程正在占用 CPU 时间时，内核减少其优先级。由于就绪进程的优先级最终会升到足够高而被调度，这种策略会防止某些进程的饿死现象。

在内核中运行的进程由于阻塞于资源或事件而放弃 CPU。当它再次变为就绪后，它被赋

予内核优先级。内核优先级高于任何用户优先级。传统 UNIX 内核中，调度优先级的值范围是 0～127 的整数，值越小，优先级越高。(由于 UNIX 系统绝大部分用 C 写成，它遵循标准习惯从 0 开始计数)。例如，4.3BSD 中内核优先级为 0～49，用户优先级 50～127。用户优先级随 CPU 使用的不同而变化，而内核优先级是固定的，因睡眠原因而定。正因为如此，内核优先级也称为睡眠优先级。表 2-2 是 4.3BSD UNIX 的睡眠优先级。

表 2-2 4.3BSD UNIX 中的睡眼优先级

表 2-2 4.3BSD UNIX 中的睡眼优先级

优先级	值	描述
PSWP	0	对换守护进程
PSWP+1	1	分页守护进程
PSWP+1/2/4	1/2/4	其他内存管理活动
PINOD	10	等待释放 i 节点
PRIBIO	20	等待磁盘 I/O
PRIBIO+11	21	等待释放缓冲区
PZERO	25	优先级基线
TTIPRI	28	等待终端输入
TTOPRI	29	等待终端输出
PWAIT	30	等待子进程结束
PLOCK	35	非强制资源锁等待
PSLEP	40	等待信号

第 5 章对 UNIX 调度器详细讨论。

2.7 信号

UNIX 使用信号(signal)通知进程异步事件和处理异常。例如，当用户在终端上键入 control-C 后，系统就会向前台进程发送 SIGINT 信号。与此相仿，进程终止时,SIGCHLD 信号被发送给它的父进程。UNIX 定义了许多信号(4.3BSD 和 SVR3 中都有 31 个)。大多数信号是保留为特定目的的，只有两个(SIGUSR1 和 SIGUSR2)可按应用程序所需使用。

信号产生有许多种方法。进程可用系统调用 kill 直接向一个或多个进程发送信号。终端驱动程序响应某些击键和事件,向与它相联的进程发送信号。内核产生信号通知进程硬件异常，或超过定额之类的事件。

每个信号有一个缺省的响应动作，一般是进程终止。某些信号缺省为忽略，有些则会挂起进程。进程可以使用 signal(System V)，sigvec(BSD)或 sigaction(POSIX.1)指定替代缺动作的其他处理动作。这个处理动作可以是用户指定的信号处理程序，也可以是忽略信号，甚至可以回到缺省设置。进程可以选择临时阻塞信号；这样信号只有当阻塞取消时才被发送给进程。

进程不是立即响应信号。当信号产生时，内核通过在 proc 结构的挂起信号掩码设置一位来通知进程信号发生。进程必须在知晓这个信号后响应它，而这只有当它被调度后才进行。一旦进程开始运行，它将在返回正常用户处理前处理所有挂起的信号。(这并不包括运行于用户态的信号处理程序本身)。

当信号发生的进程恰处于睡眠状态该怎么办呢？是信号挂起直到进程被唤醒,还是中断睡眠？答案取决于进程睡眠的原因。若进程睡眠是等待某一很快将会发生的事件，如磁盘 I/O 完成，就没必要唤醒进程。而另一方面进程等待的可能是诸如终端输人这类事件，进程

无法知道它会阻塞多久。这种情况下，内核中断进程睡眠，中止进程正在阻塞中的系统调用。4.3BSD 提供 signinterrupt 系统调用控制信号应如何影响系统调用处理方式。用户可以使用 siginterrupt 指定被信号中断的系统调用是应中止还是重新启动。第 4 章会更详尽地讨论信号这个主题。

2.8 新进程和程序

UNIX 是一个多道程序环境，系统中任何时刻都有若干进程活动。在某一时刻进程运行一
单一程序，许多程序可以并发的运行同一程序。这些进程共享内存程序正文的单一副本，但每个进程有自己单独的数据和堆栈区。而且一个进程在任何时刻可以执行新的程序，并可以在它的整个生命周期中运行几个程序。由此 UNIX 将进程和其运行的程序严格地区分开来。

为了支持这样的环境，UNIX 提供了几个系统调用创建和终止进程，以及执行新程序。fork 和 vfork 系统调用创建新进程，exec 调用执行一个新程序。exit 系统调用终止进程。注意进程也可因收到信号而终止。

2.8.1 fork 和 exec

fork 系统调用创建一个新进程。调用 fork 的进程称为父进程，新进程是子进程，父子关系构成了如图 2-4 所示的进程层次结构。子进程几乎就是父进程的完全复制。它的地址空间是父进程的的复制，开始也运行的是同一程序。实际上，子进程是从 fork 返回开始用户态执行的。

由于父进程和子进程都是从 fork 返回并继续执行同一程序，需要一种方式区分它们，并使它们能照此运行。否则，不同进程将不可能做不同事。出于这种考虑，fork 系统调用为父子进程返回不同的值——子进程中 fork 返回 0，父进程中返回子进程的 PID。

许多情况下，子进程从 fork 返回后很快会调用 exec 来开始执行新程序。(库函数提供几种不同形式的 exec，如 exece，execve 和 execvp。每个使用一组稍有不同的参数，预处理后调用同一系统调用。这个一般名称 exec 可指代这组函数中的任一函数。下面例子 2-2 中就是 fork 和 exec 的常用组合。

实例 2-2 使用 fork 和 exec

```
  if((result = fork()) == 0){
    /* child code */
              if(execv( ”new_program”, …)< 0)
                      perror("execve failed");
                      exit(1);
  }else if(result ＜ 0){
              perror( ”fork”); /* fork failed */
  )
/* parentt continues here */
...
```

由于 exec 用新程序覆盖已有的进程，子进程不会返回旧程序.除非 exec 失败。exec 调用成功后，子进程的地址空间替换为新程序的地址空间，子进程返回后程序计数器中将是新程序的第一条可执行指令。

由于 fork 和 exec 经常被组合使用，有人会考虑是否使用一个系统调用来完成两个任务，创建一个执行新程序的新进程。较早的 UNIX 系统[Thom 78]为子进程复制父进程地址空间也造成很大负担(在 fork 执行时)，而子进程却仅仅将其遗弃，并替换为新进程。

分开这两个调用有很多好处。在许多客户-服务器应用中.服务器程序可能会 fork 许多

执行同一程序的进程⑨。相反，有时程序只需执行新程序而无需创建新进程。最后，在 fork 和 exec 之间，子进程可以有选择地执行一系列任务以确保新程序以所希望的状态运行。这些
任务包括：

·重定向标准输入，输出或错误。

·关闭从父进程继承来的而新程序并不需要的打开文件。

·改变 UID 或进程组。

·重置信号处理程序。

若单一系统调用试图完成所有这些功能将是笨重而低效的。现有的 fork-exec 框架提供了更强的灵活性，清晰而且模块化强。在 2.8.3 节我们将考虑减少这种划分对性能造成的影响。

2.8.2 进程创建

fork 系统调时创建的进程几乎就是父进程的精确复制。唯一的不同是用来区分二者的部分。从 fork 返回后，父子进程执行同样的程序，有同样的数据和堆栈区，并从紧跟 fork 后的指令继续执行。fork 系统调用必须执行如下操作；

l.为子进程的数据和堆栈区保留交换空间。

2.为子进程分配新的 PID 和 proc 结构。

3.初始化子进程的 proc 结构。某些域(如用户和组 ID，进程组和信号掩码)直接从父进程复制，某些设为 0(驻留时间，CPU 使用率，睡眠渠道等等)，其他(如 PID，父 PID，指向父进程 proc 结构的指针)初始化为子进程特有值。

4.为子进程分配地址转换表。

5.分配子进程的 u 区，并从父进程复制内容。

6.更新 u 区，使其指向新的地址映射表和交换区。

7.将于进程加入父进程执行的共亨正文段的进程组。

8.以一次一页的方式复制父进程地址数据和堆被区，并更新子地址转换表，指向这些新页面。

9.取得被子进程继承的共享资源的指针，如打开的文件和当前工作目录。

10.通过复制父进程寄存器的快照初始化子进程的硬件上下文。

11.使子进程就绪，并加入调度队列。

12.设置子进程从 fork 返回 0 值。

13.向父进程返回子进程的 PID。

2.8.3 fork 优化

fork 系统调用必须给子进程提供一个与父进程地址空间逻辑上明显不同的副本。大多数情况下，当子进程在 fork 后调用 exec 或 exit，它会遗弃这个地址空间。因此，在老 UNIX 系统的实现中，为进程建立一个实际的地址空间副本是非常浪费的。

这个问题通过两种方式来解决。第一种是 copy-on-write 方法，它首先由 System V 采用，现在为大多数 UNIX 系统使用。在这种方法中，父进程的数据和堆栈页暂时设置为只读，并被标记为 copy-on-write。子进程有一份自己的地址转换表，与同父进程共亨内存页。而无论是父进程还是子进程试图修改页时,会产生页面失效异常(由于页面标为只读)并且会调用内核的失效处理程序。处理程序识别这是一个 copy-on-write 页，为其建立一个新的可写的页副本。由此只有那些被修改的页才需复制，而不是全部页面。若子进程调用 exec 或 exit，页面回到原来的保护方式，copy-on-write 标识被清除。

BSD 提供了另一种解决方案-- 新的 vfork 系统调用。若用户进程希望在调用 fork 后

随即调用 exec，它可以调用 vfork 而不是 fork。 vfork 不进行复制。相反，父进程将自己的地址空间租借给子进程，并将自己阻塞，直至于进程将地址空间还给它。因此子进程会一直使用父进程的地址空间直到它调用 exec 或 exit。于是内核将地址空间返回给父进程并唤醒它。

vfork 非常快，甚至连地址映射表也无需复制。地址空间传给子进程只是简单地拷贝地址映射寄存器。然而，由干它允许一个进程使用并修改另一进程的地址空间，这是一个非常危险的调用。某些程序，如 csh，就利用这个特性。

2.8.4 执行一个新程序

exec 系统调用用新程序覆盖调用它的进程的地址空间。若进程由 vfork 创建，exec 将原地址空间返回给父进程。否则，释放原地址空间。exec 给进程一个新的地址空间，并加载新程序的内容。当 exec()返回，进程从新程序的第一条指令恢复执行。

进程地址空间包括几个不同的组成部分⑩：

·正文(Text)包括可执行代码，对应于程序的正文区。

·初始化数据 包括在程序中显式初始化的数据对象，对应于可执行文件中的初始化数据区。

·未初始化数据 历史上称为静态存储块(bSS)，包括程序中声明但未初始化的数据变量。在这一区域的对象在第一次访问时初值为 0。在可执行程序中包括几个都为 0 的页是非常浪费的，程序头中只是简单地记录了这个区域的大小，而由操作系统在地址空间中产生全 0 的页面。

·共享内存 许多 UNIX 系统允许进程共享内存区。

·共享库 若系统支持动态链接库，进程中可能会包含几个分离的内存区，各区中是同其他进程共享的库代码和数据。

·堆(Heap)供动态分配内存使用的内存资源。进程通过 brk 或 sbrk 系统调用从堆中分配内存，也可以使用诸如 malloc()这类的标准 C 库函数。内核提供每个进程一个堆，需要时可以扩展它。

·用户栈 内核为每个进程分配一个栈。在传统 UNIX 实现中，内核可以透明地捕获栈溢出异常，并将用户栈扩展到一个预置的最大值。

共享内存是 System V UNIX 中的标准，但在 4BSD(直到版本 4.3)没有。许多基于 BSD 的商业变体作为增值特性支持共享内存和某种形式的共享库。在下面对 exec 的描述中，我们考虑没有这些特性的简单程序。

UNIX 系统支持许多可执行文件格式。最老的是 a.out 格式,它有一个 32 位的头，紧接着是正文，数据区和符号表。程序头包括正文区，初始化数据区和未初始化数据区的大小，以及程序入口点，即程序执行的第一条指令地址。它还包括一个幻数(magic number)，用以标明文件是一个有效的可执行文件，并给出文件格式的更详细信息，例如文件是否是分页式，或数据区是否开始于页边界等等。每个 UNIX 变体都定义一组它所支持的幻数。

exec 系统调用必须执行下述任务；

l.分析路径名并访问可执行文件。

2.验证调用者有执行该文件的权限。

3.读文件头并检查它是合法可执行的(11)。

4.若文件的模式中有 SUID 或 SGID 位标识,把调用者的有效 UID 或 GID 分别改变为这个文件的属主的 UID 和 GID。

5.将 exec 的参数和环境变量复制到内核空间，当前用户空间将当被删除。

6.为数据和堆栈区分配交换空间。

7. 释放旧的地址空间及相关的交换空间。若进程由 vfork 创建，将老空间返回父进程。

8.为新的正文，数据和堆栈分配地址转换表。

9.设置新的地址空间。若正文区已经处于活动状态(某些进程已在运行同一程序)，那么就共享它。否则，从可执行文件中读取它。UNIX 进程一般是分页式，即只有当程序需要时，才将页面读入内存。

10.将参数和环境变量复制到新的用户找上。

11.由于原处理函数不在新程序中，将所有信号处理函数重置为缺省动作。在调用 exec 前被忽略或阻塞的信号仍忽略或阻塞。

12.初始化硬件上下文。大多数寄存器设为 0，计数器设置为程序入口点。

2.8.5 进程终止

内核中的 exit()函数终止进程。当进程被信号杀死时，由内核内部调用这个函数。程序也可以调用 exit 系统调用，它会调用 exit〔〕函数。exit()函数执行下列动作；

1.关闭所有信号。

2.关闭所有打开文件。

3.释放正文文件和如当前日录之类的资源。

4.写登记日志。

5.proc 结构中保存资源使用统计和退出状态。

6.将状态置为 SZOMB(僵尸)，并把 proc 结构加入到僵尸进程链表中。

7.init 进程继承已退出进程的所有活动子进程〔使 init 成为它们的父进程)。

8.释放地址空间，u 区，地址转换映射表和交换空问。

9.用 SIGCHLD 信号向父进程发通知，这个信号缺省处理为忽略，因此只有当父进程希望知道子进程死亡时信号才发生作用。

10.若父进程在睡眠，唤醒它。

11.最后，调用 swtch()调度一个新进程运行。

当 exit 完成，进程处上僵尸状态。exit 不释放父进程的 proc 结构，因为它的父进程可能会访问退出状态和资源使用信息等。父进程负责释放子进程的 proc 结构，下面将介绍这种过程。当那些操作完成后，proc 结构返回到自由链表，清除操作完成。

2.8.6 等待进程终止

一个进程经常需要知道子进程何时终止。例如，当 shell 派生出一个前台进程执行命令，它必须等待子进程结束后，再提示你输入下一条命令。当后台进程终止时，shell 可能通过输出一个信息通知用户。shell 检索子进程的退出状态，以便用户根据其成功和失败的情况采取不同的动作。UNIX 系统提供下列凋用等待进程终止；

```
wait(stat_loc);                          /* System V, BSD, and POSIX.1 */
wait3(statusp, options, rusagep);        /* BSD */
waitpid(pid, stat_loc, options);         /* POSIX.1 */
waitid(idtype, id, intop, options);      /* SVR4 */
```

wait 系统凋用可以使进程等待子进程终止。由于子进程的终止可能早于 wait 调用,wait 也必须处理这种情况。wait 首先检查调用者是否有死亡或挂起的子进程。若有立即返回，若没有死亡的子进程，wait 阻塞调用者，等待它的一个子进程终止，并立即返回。在上述两种情况下，wait 返回死亡子进程的 PID，将子进程的退出状态写入 stat_loc，并释放它的 proc 结构(若多于一个进程死亡,wait 只对其找到的第一个做处理)。若子进程正被跟踪，在子进程收到信号时 wait 也返回。若调用者没有子进程(死的或是活的)，或 wait 被信号中断，wait 返回错误信息。

4.3BSD 提供 wait3 调用(因其有 3 个参数而得名)，它也返回子进程的资源使用信息(子进程的用户和系统时间，以及所有死亡于进程)。POSIX.入标准［IEEE 90］增加了 waitpid 调用。它使用户 作参数等待特定进程 ID 或进程组的子进程。wait3 和 waitpid 支持两个选项：WNOHANG 和 WUNTRACED。WNOHANG 会使调用在没有死亡子进程的情况下立即返回。WUNTRACED 在子进程处于挂起或恢复执行时也返回。SVR4 的 waitid 调用提供上述所有特性的一个超集，它允许用户指定所等待的进行 ID 或组，以及要捕捉的特定事件，并且返回更详尽的子进程信息。

2.8.7 僵尸(Zombie)进程

进程退出后，在父进程清除它以前，它一直处于僵尸僵尸状态。在这个状态下，它所保留的唯一资源就是 proc 结构，其中包括退出状态和资源使用信息(13)。这些信息可能对父进程有用。

父进程通过调用 wait 获取这个信息，wait 也释放 proc 结构。若父进程先于子进程死亡，init 将继承它的子进程。当这些子进程死亡时，init 调用 wait 释放子进程的 proc 结构。

当一个进程先于父进程死亡而父进程又没有调用 wait 时会产生一个问题。子进程的 proc 结构将不会释放，子进程会一直保持僵尸状态直到系统重启。这种情况很少发生，因为 shell 尽一切可能避免这种情况。然而，它会发生厂一个粗心大意写成的应用程序，它没有等待它的子进程。这种情况非常令人讨厌，因为这种僵尸将可以在 ps 输出中看见(同时用户很着急的发现它们不能被杀死--他们已经死亡)。更重要的是，它们占用 proc 结构，因而会减少处于活动状态的进程总数。

某些较新的 UNIX 变体允许进程指定它将不等待其子进程。例如，在 SVR4 中，进程可以在 sigaction 系统调用中设定 SA_NOCLDWAIT 标志，以便为 SIGCHLD 信号指定处理。这请求内核在调用者的子进程终止时不创建僵尸状态。

2.9 小结

我们已经描述了传统 UNIX 内核中内核和用户进程的交互。这使我们对系统有了一个更为全面的了解，今后我 们就可以针对系统的特定部分详细讨论。像 SVR4 和 Solaris 2.x 这样的现代 UNIX 变体引人了一些新的特性，在后续章节中我们将做详细介绍。

2.10 练　习

1．当处理(a)上下文切换,(b)中断或(c)系统调用时，哪些信息是内核要明确保存的？

2．动态分配对象 proc 结构和描述符表块的优点是什么？缺点是什么？

3．内核是如何知道进行哪个系统调用的？它是如何访问调用参数的(在用户指定)？

4．比较系统调用和异常处理的相同和不同点？

5．许多 UNIX 系统通过为其他版本的系统调用提供用户库函数的方法与其他版本 UNIX 兼容，为什么应用开发人员关心函数系统实现是通过库还是系统调用？

6．当选择是在用户库中还是作为系统调用新一个函数，什么问题是库程序员必须考虑的？当库必须用多个系统调用实现函数又会怎样，

7．为什么限制中断处理程序的工作量非常重要？

8．假设系统有 n 个不同的中断优先级，在某一时刻所能嵌套的中断次数最多是多少？

9．Intel 80x86 不支持中断优先级。它提供两个指令用于中断管理一 CLI 屏蔽中断，STI 使中断。写一个算法在这种机器上的软件中实现中断优先级。

10．当资源可用时，wakeup()例程唤醒所有阻塞于它的进程。这种方法的缺点是什么？还有其他的方法吗？

11.提出一种合并 fork 和 exec 函数的系统调用？定义它的接口和语义。它应如何支持诸如 I/O 重定向，前台或后台执行和管道等特性？

12.从 exec 系统调用中返回错误会有什么问题？内核应如何处理这个问题？

13.在你所选择的 UNIX 系统，写一个函数允许一个进程等待它的父进程终止。

14.假设进程直到它的子进程终止才阻塞，它如何确保当子进程终止以后被清除？

15.为什么终止进程唤醒它的父进程？

2.11 参考文献

①有些 UNIX 系统不使用虚存系统。这些系统包括早期的 UNIX 发行(第一个虚存系统是在 20 世纪 70 年代出现的)——见 1.1.4 节和某些实时 UNIX 变体，本书只针对有虚拟内存的 UNIX 系统。

②UNIX 系统在某些情况下走的久远；例如，它将磁带驱动器当做是字符流，这使得应用程序处理错误和一些特殊情况非常困难。磁带接口本质上是基于记录的，并不能非常好的和 UNIX 设备框架匹配[Allm 87]。

③系统在内核态时也可以发生中断.此时，系统在处理完中断后仍保持内核态。

④SVR3 为了实现进程跟踪(见 6.2.4 节)的目的还特设一个以 stopped 状态，当一个被跟踪的进程接收到任何信号后，它进人 stopped 状态，然后内核唤醒它的父进程。

⑤或者是最低端，在某些机器上堆栈是向下生长的。在某些系统中，堆钱指针指向的是可入栈的下一个元素的地址。

⑥像 SVR4 这样的现代 UNIX 系统将用户信用信息保存在动态分配的，引用计数的数据结构中，同时在 proc 结构中保存一个指针。8.10.7 节将详细讨论这种结构。

⑦ 某些处理器，加 Intel 80x86，不能从硬件上支持中断优先级。在这种系统中，照作系统必须用软件来实现 ipl。练习里更详细地讨论了这个问题。

⑧近期的 UNIX 版本中若干 wakeup()的替代函数，如 wake_one()和 wakeprocs()。

⑨ 在现代多线程 UNIX 系统中就没必要这么做了 一服务器只需简单地创建一系列线程。

⑩ 这种计割在功能上是很自然的，内核并不识别这么多的部分。例如，SVR4 将地址空间视为一系列共享和私有映射的集合。

(11) 当首行为；

```
#!shell-nameexec
```

也可以启动 shell 脚本程序。此时，exec 调用由 shell-name 指定的程序(通常是一个 shell 程序的名字，但事实上它可以是任何可执行的文件)并将脚本的名字作为第一个参数传递给它。 某些系统(如 UnixWare)要在 shell-name 前加一个空格。

(12) SVR4 通过库函数支持 wait3 和 waitpid。

(13) 某些实现通过一个 zombie 结构来保存这些数据。