虞育新等编

UNIX

设备驱动程序

北京科海培训中心

UNIX 设备驱动程序

虞育新　等编

北京科海培训中心

目　录

第一章　UNIX 及其 I／O 子系统

本章介绍了 UNIX 操作系统的概况以及它实现 I／O 操作的方法。其中的一些内容只用于引起兴趣—编写设备驱动程序不需要有关这些内容的细节；另外的内容则以主题为中心，但在初步的讨论中不提供完整的细节。当使用不精确或非直观的描述时，讨论的内容对设备驱动程序或不是重要的，或者将在以后章节中更精确地介绍。

自从七十年代早期，UNIX 操作系统已经历了极大的发展，在它诞生的贝尔实验室(以及后来的 AT&T 信息系统）里，以及在采用 UNIX 作为软件系统基础的许多大学和公司里。但派生的两种 UNIX 系统在内部实现上有很大的不同。关于 UNIX 系统内部操作的具体实现已超出本书的范围，但设备驱动程序接口对于大多数 UNIX 版本在概念上是类似的。

· UNIX 操作系统 ·

用最一般的术语，UNIX 软件系统由下列三部分组成：

1. UNIX 内核
2. 大量的用户进程
3. 存于外存中的数据文件系统

内核是系统的心脏，因只有该程序能直接存取和控制系统硬件，包括处理器、主存、以及 I／O 设备。用户进程对应于命令，在终端上键入并为用户完成任务。内核的功能是给这些进程提供对系统资源的存取，包括文件系统。它类似于一个协调者，允许硬件和软件系统实现多用户，多任务，系统分时。

对于一个从交互式终端检查活动 UNIX 系统的用户内核是不可见的。所有能运行的程序，包括编译器，I／O 实用程序和终端命令解释程序都是用户进程。对一个编写应用程序的用户内核以系统调用方式可见。这类似于一般的函数或子过程，但它们必须由系统控制和协调才能完成操作。

完成特定任务的用户进程要求共享系统资源。通常有许多活动的用户进程竞争资源。内核从全局上进行管理，根据优先权决定进行服务的次序。

内核

UNIX 内核包括一个大的可执行二进制映象。其中的大多数模块用 C 语言书写，少量用汇编语言书写。这些模块按标准的 UNIX 程序开发工具编译和汇编，然后连接为一个二进制映象。它被装入到主存中，其执行由一个初始化过程启动，这些细节我们不作介绍。

内核完成三个基本工作：

- 创建用户进程并对它们的执行进行调度
- 提供系统服务
- 处理硬件中断的意外事件

内核的逻辑概述如下：

- 维护已存在用户进程表并为每个进程相关一个调度优先权。
- 检查该进程表，分离就绪和非就绪进程。一个进程可由于许多原因处于非就绪状态，最一般的就是等待 I／O 操作结束以及等待另外进程完成执行。也可能是等待主存等资源为可用。
- 从就绪进程中选择最高优先权的进程运行。该进程控制 CPU，并且内核的执行暂时挂起。
- 选中的进程运行到产生一个系统调用，这时用户进程挂起，同时激活内核。内核处理系统调用并重新选择一个进程执行。

图 1-1 列出一个简化的进程表。进程 1，40 和 41 为可运行的（R），进程 20 和 24 处于睡眠状态（S）并等待特定的事件。进程 24 等待从一个设备读，进程 20 等待一个子进程退出。

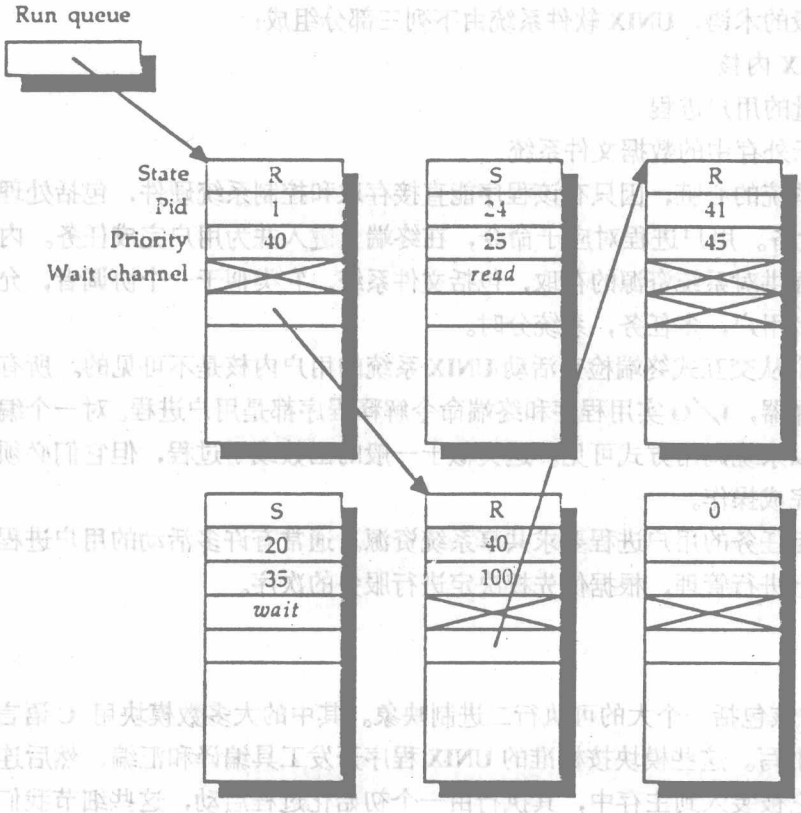

图 1-1　进程表项

一些系统调用可由内核中的执行代码完全满足。例如 lseek 将文件指针设置于要求处。

对圆形设备进行 I／O 的要求则常需要一系统的动作。这时内存异步地启动设备操作，但并不等待其结束。要求服务的进程被标记为未就绪并保持该状态到一个设备中断，以指出 I／O 操作已完成。

用户进程

在 UNIX 系统上运行的应用程序源代码被编译并连接为一个二进制文件，即可执行映象。一个用户进程即 UNIX 内核中一个映象的执行。

用户进程执行一个映象时，将共享指令（正文段）装入内存的只读区。（内核而不是硬件使得该区只读。）另外的两个内存段定位成用户读／写数据区域，第一个用于命名的变量，第二个用于堆栈。内核作另外的一些准备后将适当的值装入硬件程序计数器以启动该进程。

某一时刻进程的状态完整描述包括以下项：

1. 内存空间，包括代码和数据区域。
2. CPU 通用寄存器和程序计数器（硬件内容）的值。
3. 其它信息，如打开的文件，当前工作目录，以及系统调用请求的状态（软件内容）。

软件内容信息保存在由内核维护的数据结构中。

执行一个进程时其代码和数据区位于主存中，其寄存器和程序计数器值装入到 CPU 寄存器中。若进程必须挂起，寄存器和程序计数器值被复制到内存的数据结构中，因而可以由另外进程使用 CUP。

只要挂起进程保留在主存中，它就可以通过恢复这些值来重启。但内核可能会将进程映象送到外存（盘）上，以释放主存供其它用户使用。此时仅有很少量的信息保留在内存中—仅够允许内核在重启该进程时所需。

用户进程运行于伪计算机（由 UNIX 内核组成但不等同于真正的系统硬件）上时，伪计算机执行一些额外的命令（系统调用），但不允许对机器的所有部分进行存取，如设备中断和优先权指令。进程挂起时，传送到外存上，以后再重启，这些事件完全不被进程可见。对用户进程，内存中代码和数据区属于内核，且其它用户进程都是不可见的。

系统和用户地址空间

RTU 是一个命令分页虚存系统。每个用户进程都有自己的地址空间，开始为 0 或更大的值（16 兆或更大），终值基于处理器类型和系统中可用的外存数量。UNIX 内核有它逻辑上不同的地址空间，分别称为用户和系统虚拟地址空间。对 MC68000 系统，系统虚拟地址空间完全与用户虚拟地址空间分开，而对 MC68020 系统，用户虚拟地址是系统虚拟地址空间的子集。

MASSCOMP 硬件和操作系统将用户和系统虚拟地址空间映射为大小等于 4096 字节的物理存贮页。从 0 开始连续的用户进程虚地址空间映射为分散的非连续物理存贮页。而且，用户进程地址空间的单个页可能会被移到内核的外存，用以释放主存空间供其它进程使用。见图 1-2。

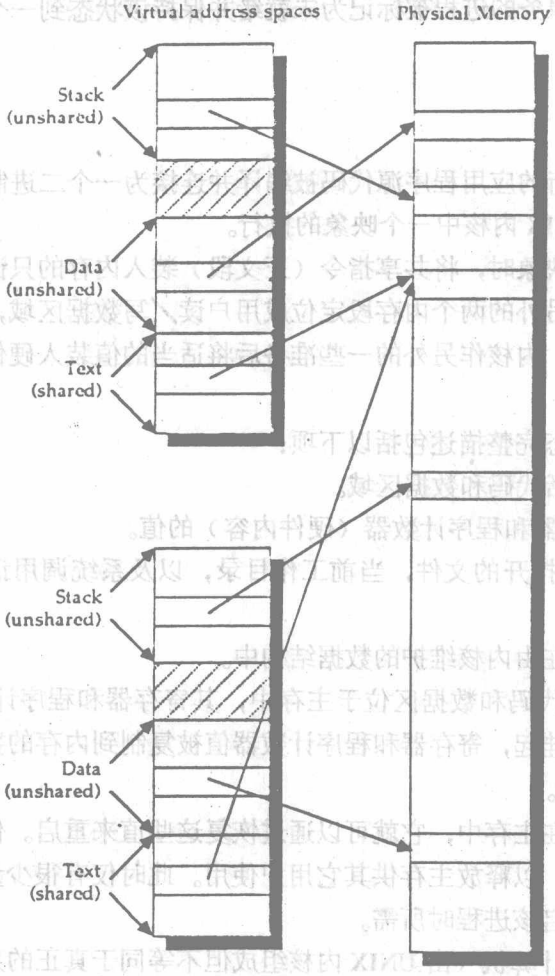

图 1-2　虚拟地址空间和物理地址空间的映射

一般用户进程的设计者通常不需要注意虚—实地址转换，因为内核和硬件将自动处理。但设备驱动程序的设计者必须考虑这些地址问题。驱动程序处理用户进程地址，系统（内核）地址，以及 I／O 总线地址。本书详细说明设备驱动程序碰到的每一情况相对应的地址类型，还描述了驱动程序调用的内核功能，用于将数据从一个地址空间复制到另一地址空间，以及不经过复制使数据出现于另一地址空间（这是通过处理许多地址映象表来实现的）。

• UNIX 的用户窗口 •

UNIX 系统最初被设计和制造成分时系统。许多用户同时通过终端与 UNIX 联结，用

键盘输入，通过显示屏幕或监视器显示字符。最近的 UNIX 版本增加了许多强有力的用户接口特性，如多窗口，使用菜单执行操作，或用点设备选择偶像来执行操作，绝大多数的 UNIX 软件工作于终端。

用户使用可能有任意个参数的一系统命令。尽管根据约定，以短线开始的参数是标志或选项，它们影响命令的执行，但一个命令的参数常为要处理的一个或多个文件。参数以正文串传送给命令。

使用文件参数时 UNIX 的多数命令是灵活的：若不指定输入文件参数，则从键盘输入；若不指定输出文件参数，则在终端屏幕上输出。如 Cat 命令（con-catenate 的缩写）复制任意数目的文件到终端：

```
cat preamble data
```

先复制 preamble 到终端，然后复制 data 到终端。

通常情况下从键盘输入或以终端输出的命令可以对输入／输出重定向。如 ls 命令列出目录中的所有文件：

```
ls -l
```

它打出当前目录中所有文件的长格式信息：

```
-rw-rw-rw-    1  fred user 101 Jun 4 14:55 preamble
-rw-rw-rw     1  fred user 5428 Jun 4 15:13 data
```

上述信息可用下述命令写入一个文件 dir 中：

```
ls -l > dir
```

类似地，命令 WC 也计数一个文件中的字数，行数和字符数。指定 WC 的文件参数时，WC 也计算它的字数，行数和字符数该输入可由下面的命令从 data 文件重定位输入：

```
wc < data
```

最后，一个命令的输出可通过管道 pipe ，程序间发送数据的特殊机构，来作为另一命令的输入。一个 pipe 类似于程序的一个文件。命令：

```
cat preamble data | wc
```

用于计算文件 preamble 和 data 中的总共字数，行数和字符数，而不需建立临时文件。UNIX 系统中的命令行常被称为管道线，因为常包含以管道连接的一些命令。这些命令称为过滤器，因为它们处理流经管道的数据。

· UNIX 的程序员窗口 ·

UNIX 系统的程序开始执行时将命令参数放在内存中保留给子程序调用堆栈的区域。在高级语言中，命令参数总是给程序的“main”过程。如果不是这样，则特殊的库函数用以存取这些命令参数。参数只是正文串，尽管命令行的每个“字”作为单独的参数传递。多数情况下，命令参数不被 shell 改变，但必要时 shell 扩展文件名的匹配符为尽可能多的参数。 shell 还有专门用于变量替换的符号，同时引号用于禁止参数的改变。I／O 重定向也由 shell 处理，将在下面详细说明。因此，即使 UNIX 中最简单的程序也可以毫不费力地由程序员“实现”这些功能。

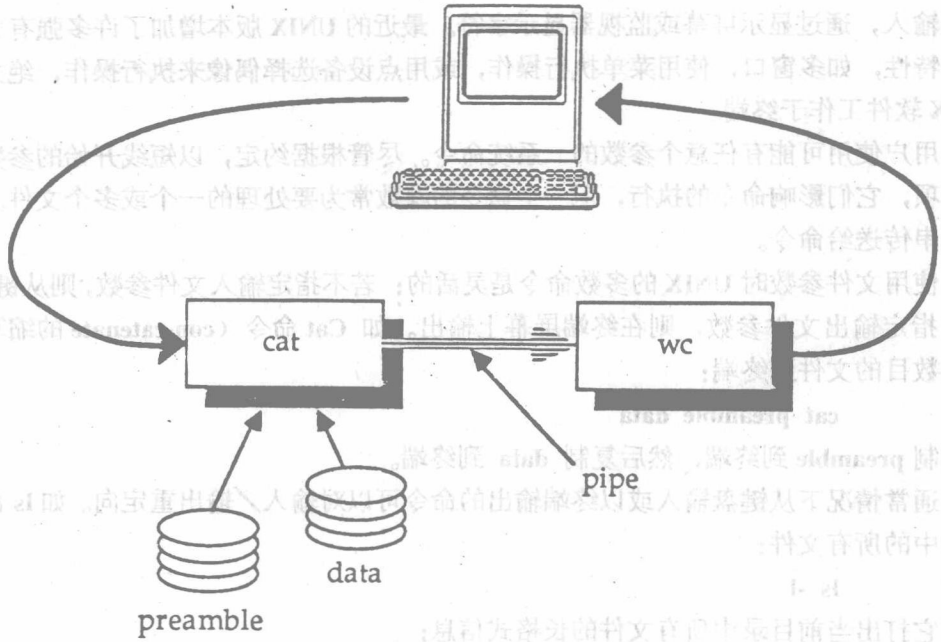

图 1-3　管道线执行

UNIX 实现了大量的系统调用。系统调用提供的服务种类有：

- I／O 操作。提供存取共享 I／O 设备和描述其状态的全局数据结构。打开、关闭文件和设备，读、写数据，设置设备状态，以及读和写系统数据结构。
- 进程控制。允许进程控制自身的执行。进程可以分配内存，设置调度优先权和其它参数，锁定其存贮，装入并执行一个新程序，以及等待事件。还能创建新进程。
- 进程间通讯。允许进程向另一进程发送信息。有许多种方法用于进程间发送信息：signal、管道、共享内存，消息队列和 semaphore。除此外 RTU 实现异步系统陷入（trap）AST，它在许多方面类似于 signal，但少了 signal 的很多限制。
- 定时服务。进程可使用系统的时间和时钟间隔来同步其活动。
- 状态信息。有大量的调用返回进程及其子进程、文件系统和 I／O 设备的状态信息。

用高级语言编程时，系统调用象特殊的库调用。因为许多系统调用返回某类的值，这些调用一般保存一个非法的值集来说明发生了一个错误。一个特殊变量用于存贮出错码，说明错误。

I／O 系统调用

一般，一个文件在被使用前必须先打开，使操作系统初始化各种内部数据结构，以允许更高效地进行后续的 I／O 操作。打开文件后，返回一个文件号，它是用以指明该文件的一个小整数。其它的 I／O 操作以文件号为参数，而不是以文件名为参数，因此该文件名只须检查一次。

有两个打开文件的系统调用。它们是：

fd=open（file, mode）

这里 file 是指定文件路径名的字符串， mode 包含打开该文件后所允许操作的各种标志位。 mode 最常见值为对一已存在文件只读、只写、或读和写。

另一打开文件，用于建立一个新文件或覆盖一个已存在文件的系统调用为：

fd=creat（file, protection）

file 是要创建文件的路径名字符串。protection 是一个数，说明可以打开作各种类型存取的用户。注意许多 UNIX 的最近版允许 open 建立文件时指定适当的方式位，并将 protection 作为一个附加的参数。但 creat 与已存在的程序兼容。

一旦文件打开，主要的操作是读或写。系统调用：

actual=read（fd, address, count）

从 fd 读取最多 count 字节到 address 地址。read 返回实际传输的字节数，该值小于 0 时出错，等于 0 时说明已到文件结束，而不是出错，否则实际读取的字节数一般等于要求的字节数，尽管可能到达文件尾时 actual 小于 count。这在 fd 为一特殊文件（设备或管道）时也会出现。例如，终端设备驱动程序一般读取不超过一行的数据，而不管要求多少字节的数据。

系统调用：

actual = write（fd, address, count）

类似于 read 只是写数据到文件中。尽管设备驱动程序有写入少于要求字节数的选项，几乎所有程序将它解释成一个错误。

I／O 操作通常是顺序的，因此下一个 read 或 write 调用存取文件中的下一个数据块。数据可以通过先用下面的调用再进行读或写：

new_location=lseek （fd, offset, whence）

这里， whence 说明 offset 是从文件头、文件尾、还是文件指针的当前位置偏移一定字节数。 lseek 返回文件指针的新的位置，即从文件头开始的字节数。

当一个已打开的文件不再需要时，进行关闭：

result = close （fd）

成功时 result 等于 0。通常 close 成功，除非 fd 没有打开。

一个文件关闭后，再想读或写该文件就出错。该文件号可重新使用，因而以后的 open 或 create 可能返回先前返回的值。这是很有用的，用于实现 I／O 重定向。每个程序开始执行时先打开几个文件号。对于由命令解释程序执行的程序，这些文件号是：

- 0． 标准输入文件。它通常是终端但可以用 <file 进行重定向。
- 1． 标准输出文件。通常也是终端但可以用 >file 进行重定向。
- 2． 标准错误文件。通常也是终端但可以作 I／O 重定向。I／O

重定向的方式随 shell 不同而不同，对标准的 UNIX shell 是 2>file。

一般命令解释程序先关闭这些标准文件号之一，然后再打开指定的文件来完成读或写改向。具体细节必然有些复杂，否则命令解释程序将不能读另外的命令，因为它关闭了自身的标准输入文件。为避免该问题命令解释程序先用系统调用 fork 创建一个新进程：

pid = firk （ ）

它创建一个新进程并在新旧进程中返回不同的值。新创建的进程称为子进程，原进程称为

父进程。见图 1-4。 父进程可对子进程进行一些特权控制。特别地，父进程可通过下面的系统调用等待子进程完成执行：

pid = wait （status）

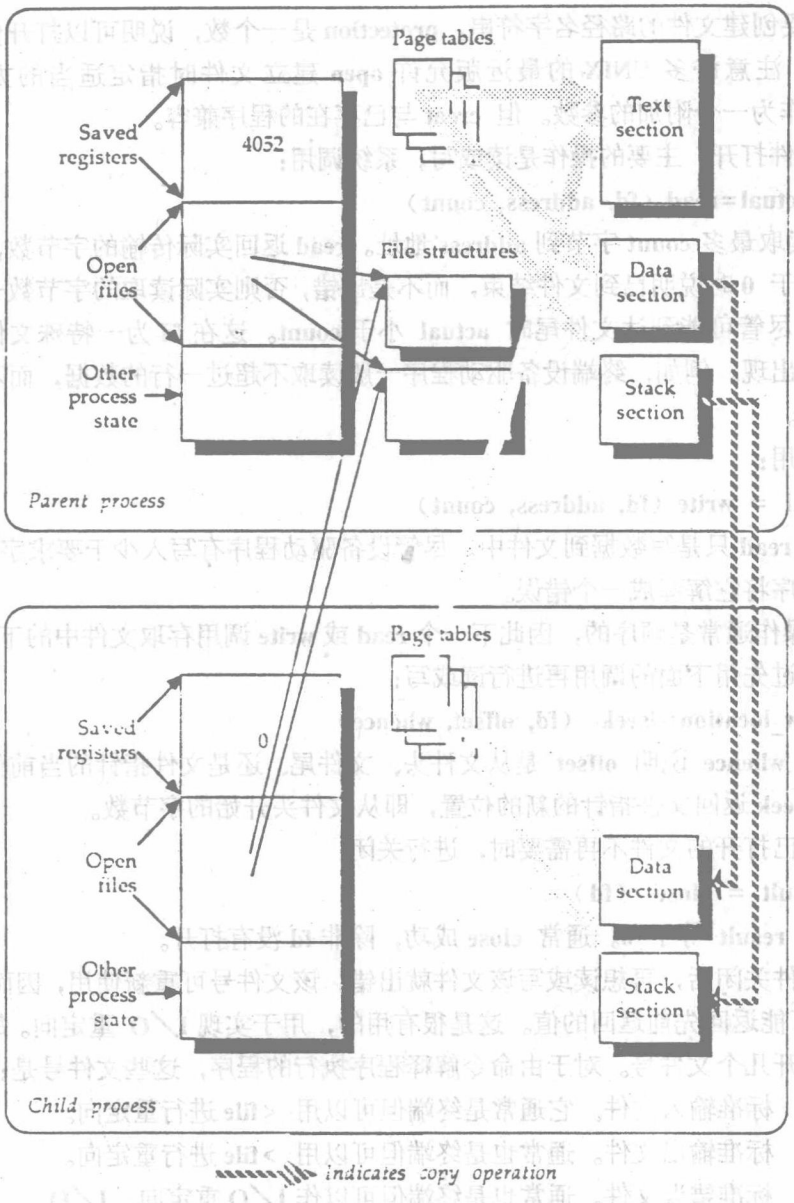

图 1-4 fork 操作

pid 指出哪个子进程完成执行。结束执行的原因返回在参数 status 中。一个进程可以是正常退出并在 status 中说明一个完成码，也可能是由于出错或从终端或别的进程接收一

个信号而结束执行。

fork 生成的子进程几乎是父进程的完全拷贝，当 pid 在父进程中是新建子进程的标识时 pid 在子进程中为 0。两个进程都可以对同一文件号进行读和写，并且开始时具有相同的存贮内容。由于子进程是父进程的复制，它可以关闭文件号以及打开新文件，而不影响父进程。每个进程都可以在不影响其它进程的情况下修改其存贮内容。

打开的文件号不是完全的复制，且继续共享相同的信息。特别地，父进程和子进程复制的文件号共享定位指针。这使得父子进程方便地对输出文件进行写，不会将其它进程写入的数据覆盖。还允许每个进程确认在数据正好读一次时取数据（假定无进程调用 lseek）。

一个程序可以对任意已打开的文件号进行复制：

new_fd =dup （old_fd）

这将复制 old_fd 到当前未用的最小数文件号。

任意进程可以执行一个新的进程：

result = exec （program, argument0, argument1, argument2,...）

这里 program 是要执行的命令或程序，argument0, argument1...是 program 的参数。约定 argument0 是程序名本身，参数表由零终止，它不是有效的字符串。

新进程替换原执行的进程。若 exec 调用成功，它并不返回，exec 调用后的语句不执行。但在过程中可以得到执行一个程序后的结果，通过创建一个新进程来执行程序并等待它退出。

命令解释程序实际使用的 I／O 改向序列为：

1. 打开指定的 I／O 改向文件。若失败，则打印错误信息并返回，等待另一命令行。
2. 调用 fork 创建新进程。
3. 父进程关闭 I／O 改向的文件号并调用 wait。
4. 子进程根据改向是标准输入，标准输出，还是标准错误以参数 0，1，或 2 调用 close。
5. 子进程用 dup 设置标准输入、标准输出或标准错误来引用用于改向的文件号。该文件号在调用 dup 后可关闭。
6. 最后，子进程执行命令行上指定的程序。

有其它与文件系统相关的系统调用，大多数我们不关心。例如，link 和 unlink 建立或去除文件名，它们不需调用设备驱动程序。类似地 stat 和 fstat 返回路径名或打开文件号的信息，它们也不调用设备驱动程序（该信息与由 ls 打印的长格式信息同样有用）。

设备驱动程序实现的一个很重要系统调用是 ioctl（用于 I／O 控制）。格式相当简单：

result =ioctl （fd, command, arg）

其中 fd 为一个已打开文件号， command 指出要求的操作， arg 说明要求的其它信息。 arg 经常是内存中控制块的地址，因而可以为 ioctl 操作提供任意数量的参数数据。

command 的解释全部属于设备驱动程序，约定不同的设备驱动程序不实现相同的命令，除非它们是同一设备类型的不同实例。例如，终端设备驱动程序的命令允许程序数据的通讯速率或改变字符如何显示；磁带设备驱动程序的命令用于磁带的反转或转到下一个文件。但若程序请求终端设备驱动程序进行反转，或请求磁带停止显示都将产生错误。

本节讨论用户进程如何存在，如何表示状态，以及如何调度。

创建和删除进程

系统初启时创建两个基本进程：

- 进程 0。交换进程 （swapper process）
- 进程 1。init 进程

只要 UNIX 系统是活动的它们就存在。 swapper 支持存贮管理操作，init 进程负责(直接或间接）启动所有其它进程。其它进程可被系统创建用户服务，例如分页。

创建这些进程后，通过调用 fork 建立用户进程。用户进程调用 fork 时内核创建几乎是调用进程完全复制的新进程。唯一的不同是新进程具有一个新的、唯一的进程标识数(PID）和一个新的父进程标识（PPID）。原进程即父进程，而新建的则为子进程。

用户进程可通过调用 exec 去替换正在执行的代码映象。因此要建立一新进程运行一新程序，已存在进程必须先调用 fork，然后调用 exec。在系统初启时 init 进程根据文件／etc／inittab 创建几个附加的进程。init 创建几个基本进程和其它进程，gettys，管理每个交互终端的登录。

一个用户连入系统时，管理终端的进程主要调用 exec 数次来运行连接序列的各个程序，最后装入 shell 或命令解释程序从终端读取命令。 shell 自身也能完成一些命令，但常须运行另外的程序（如执行已知的映象）。为此用 fork 创建新进程，然后用 exec 装入并运行指定映象。 shell 或等待该新进程的运行或继续读和执行其它命令，新进程完成工作后，调用 exit 并不再存在。 shell 接到脱机命令时也调用 exit。正等待 exit 的 init 就创建一个新进程 getty，来管理终端，并等待下一次联机。

后续的讨论说明任一时候可存在的用户进程数。每个交互终端至少有一个进程，终端 shell 不断为命令创建新的进程。这些进程可能轮流创建另外的进程。

进程定义结构—user 和 proc

内核为每个进程维护两种数据结构：

- user 。包含进程活动时需要的所有信息。当前活动进程的 user 结构命名为 u。设备驱动程序经常使用 user 结构中的许多域。
- proc 。包含进程不活动且可能交换出内存外时的信息。设备驱动程序不直接存取 proc 结构的域，但调用内核例程时常传送指向该结构的指针作为参数。

这些数据结构分别定义在 usr／include／sys／user.h 和／usr／include／sys／proc.h 中。

一个进程交换到盘上时 user 结构同时写出。proc 结构总是保存在进程表的数组里，处于主存之中。一个用户进程的状态完全定义在这两种结构中。

进程调度、分页和交换

许多用户进程包含 CPU，内存和其它硬件的使用。 RTU 内核包含用于设置这些内容的代码，通过执行调度、内存分页，和映象交换。在任一时刻可有许多进程准备运行，内核中的调度例程扫描进程表并选择就绪并具最高优先权的一个进程，该进程控制 CPU。调度程序有间隔地进行选择，最后每个用户进程都有执行时间。

因为 RTU 是分页的虚拟存贮系统，则内核和活动用户进程占有的总虚地址空间可能会超出系统的物理存贮数量。出现这种情况时内核将物理存贮中用户进程的大小 4096 字节的页复制到需要时可检索到的外存。分页由单独的内存引用驱动。不在主存的虚页引用导致一个硬件陷入。如果没有未用的物理存贮页，则将驻留在虚存的一页写到分页的文件中，以释放需要的空间。

进程 0，即 swapper，将进程映象完全移入和移出外存的文件。后续的分页及其它情况下也引起 swapper 选择一个或多个进程移出。

注：有时用户进程的部分或全部不能常驻内存。在驱动程序启动 DMA 前，它必须确认用户进程虚地址空间中的目标区域锁入物理存贮。 RTU 内核提供例程，使驱动程序可以调用以完成这种及其它相关的任务。

环境切换—当前进程

选中执行并控制 CPU 的用户进程称为当前进程。选择一个新的当前进程并开始执行称为环境切换。发生切换时， u 的内容换以一个或多个系统虚拟地址空间的页。

· 系统调用 ·

用户调用到系统调用

对应用程序员，系统调用实际上是调用子例程或函数—跳转到连入用户进程映象的一个例程。但在例程执行过程中状态发生更大的改变，进程进入内核方式并开始执行连入内核的代码。

用户模式和内核模式执行的差异是基础，前者到后者的切换通过硬件陷入或例外实现。硬件处理器处于不同的状态，在一个不同的虚拟地址空间中执行，并能执行特权指令，这称为用户进程继续执行，但正处于内核模式，从效果上看是一个内核例程。也可以说是用户进程被挂起，内核被激活，为用户进程提供服务。

当一个用户进程执行一个系统调用激活内核时，它仍是当前进程；其 user 结构占据着内核的 u 结构，位于象 read 和 write 调用的设备驱动程序上下文中。

I／O 服务

程序员编写设备驱动程序最有兴趣的系统调用是 I／O 操作。UNIX 内核提供的 I／O 服务允许用户进程将数据移入和移出硬件设备，如终端和磁盘。设备常常并不由用户进程占有或直接控制—它们的内存映象寄存器不在进程地址空间中出现。

对每种 I／O 设备，内核中都有一组例程及表格，称为设备驱动程序集。一个驱动程序可以控制几种类拟的设备，但每种设备必有一种设备驱动程序。设备驱动程序是 UNIX

系统中直接与设备联系的唯一代码。在 MASSCOMP 系统中这种联系包括读和写内存映象设备寄存器（即硬件寄存器，在软件中为内存定位）和域设备中断。另外的处理器使用特殊 I／O 指令来存取设备。

UNIX 内核包括大量用于 I／O 的代码，它们不是设备驱动程序的一部分。该代码完成与设备无关的 I／O 及服务于所有设备的文件系统。设备驱动程序的作用是为该与设备无关的 I／O 系统提供一个标准的、一致的设备接口，最大限度地去除了设备的专门特性。

除了实现实 I／O 操作（读或写数据，或设置设备状态），还有许多附加的控制服务，用于修改全局数据结构或返回状态信息。一些附加的服务可完全由与设备无关的 I／O 系统完成。设备服务的请求总是调用一个或多个设备驱动程序例程，或作出立即响应，或者是非直接的，也可能是异步的。