

# 第一章 概 论

本章首先概括 **Unix** 系统结构 然后讲述 **shell** 的含义及其有关的几个概念 并对 **shell** 命令的执行过程以及几种执行方式作了一般描述。对于一个了解多种计算机语言的读者，或许很关心编程语言的选择 本章给出了这方面的指导。如果你自信对 **shell** 有了一定的了解，可以跳过本章，但若你觉得还有些问题不太清楚，我想，阅读本章或许能使你明白许多。

## 一、 **Unix** 系统结构

**Unix** 分时操作系统自 1969 年问世以来，经学术界研究推崇，八十年代已广泛进入计算机商业市场。随着 **Unix** 影响的不断扩大，成为国际标准操作系统的呼声越来越高。它以其核心结构紧凑、功能强、良好的系统开发性、可移植性以及用户界面的灵活性 (**Shell** 等特性优势 已经发展成为各型计算机的主要操作系统。由于 **Unix** 的一系列独到而重要的设计思想和强有力的特性功能，它的出现已成为计算机界公认的操作系统发展中的重要里程碑。

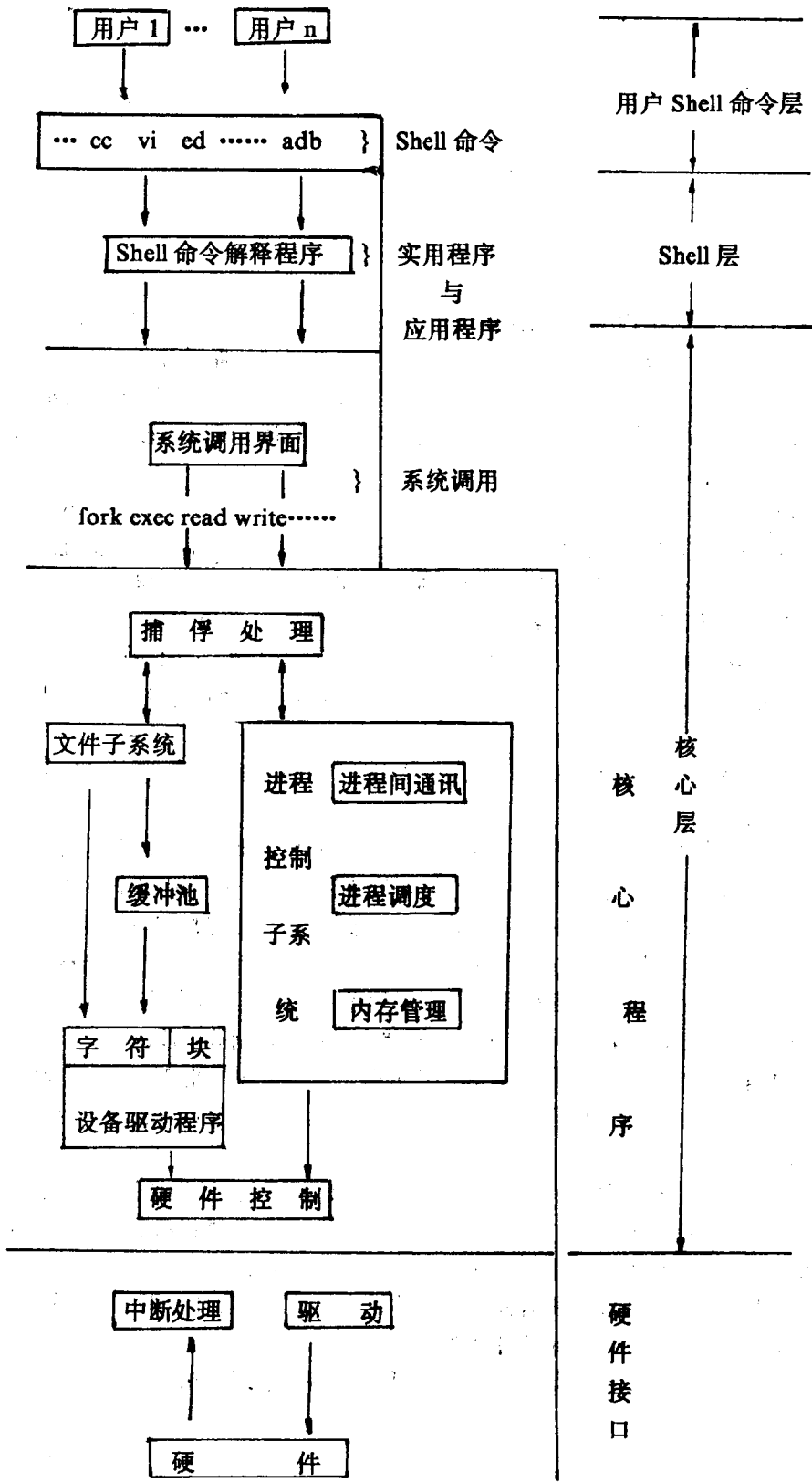
简单的讲，**Unix** 系统的整体结构分成三个层次 核心层、**Shell** 层和用户层。系统整体的层次结构图如下页所示，

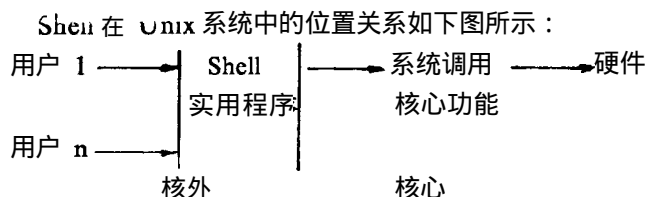
**Unix** 系统核心是常驻内存的那部分程序和数据，包括所有立即且频繁调用的系统功能程序。核心主管 **I/O** 传送、管理和控制硬件并调度进程执行。核心是整个系统的基础。所有用户态的程序不能直接访问核心里面的程序和数据，只能通过 **TRAP** 指令引入到系统调用程序。**Shell** 层是用户层与核心的界面，用户的一切要求都经由 **Shell** 与核心及机器硬件打交道。用户命令层是由一系列语言、应用程序、实用程序及其它组成的 **Shell** 命令层，提供系统管理及维护、文件目录管理、机器状态显示与控制等手段。

**Unix** 系统分时有两层含义，即进程分时和用户分时。进程分时是指在时间上把两个或多个进程交错地在一台处理机上处理。用户分时是指多个联机用户同时使用一台计算机，可以相互通信。

进程的概念可以说是 **Unix** 系统中最基本和最重要的概念之一。进程是指“程序的执行过程”是由 **fork** 系统调用建立的一个实体。除进程 0 之外，所有进程都是由其它进程执行 **fork** 创建的。进程 0 是由系统引导时建立的，它在创建了子进程 1 后变为交换进程。进程 1 即初始化进程，是系统中所有其它进程的祖先。进程 0 和 1 是两个特殊进程，前者是一个进程调度表 (**the scheduler**) 后者用于控制进程结构。

进程是一个动态的概念 具有生存周期 通常有睡眠、等待、就绪、执行、死亡等几个状态过程。





## 二、Shell 概念

1.shell 是 Unix 操作系统与终端用户之间的一种交互式命令解释程序，也是一种命令级程序设计语言解释程序。通常 shell 从一个文件（终端特别文件、用户文件或 Unix 系统文件）读取命令行、执行各种解释及置换，然后通过使用 Unix 系统原语（系统调用）来执行出结果。

shell 不属于 Unix 系统核心，但它调用了核心的大部分功能。它是用 C 语言编写的一种程序，既可以交互式地解释和执行你键入的命令，完成你与 Unix 操作系统核心的对话；又象大多数高级语言一样，定义了各种变量和参数形式，提供了强有力的条件控制结构和循环，完成你所希望的复杂的程序设计，并以并行的方式协调各个程序的执行。

2.shell 过程、shell 文本、命令文件和 shell 程序等四个概念都是类似的，区别不大，都指由命令行组成的文件。但是当文件只包含有一个或少量的几个命令语句时，通常使用术语“命令文件”而当有大量命令或使用了 shell 的循环和条件执行等功能时，通常就用术语“shell 程序”。shell 程序与批处理有一定形式上的相似，但批处理是一种非会话型的顺序执行的数据处理方式。在批处理系统中，程序由用户一次性正式提交，然后由操作系统调度执行；而 shell 象编辑程序一样是交互式程序，是 Unix 系统最重要的程序之一，是与 Unix 系统绝大多数功能调用相互作用的中心，具有比批处理更完善、功能更强的参数提供方式和条件及循环控制结构。

3.解释性语言与编译性语言是不同的，前者从源文件读取、解释然后执行；而在编译性语言中，命令行只被编译程序读取一次，进行编译，产生出一个可执行的文件，此文件可以被系统立即执行。解释性语言在运行时作解释，而编译性语言在编译时作解释。编译性语言通常产生一个比解释性语言更有效益的目标代码文件，但缺乏解释性语言那样的灵活性。此外，解释性程序比编译性程序易写易调试。

4.shell 作为一种特殊的程序设计语言，与其它高级语言比较，是有许多不同的，这主要表现在：

- shell 是解释性的，而高级语言大多是编译性的，它可以交互式地解释和执行命令，并给出提示信息。
- shell 处理的数据对象通常是文件或字符流、命令语句；而高级语言通常处理数字或字符，它具有较丰富的数据处理类型和数据结构。
- shell 使用环境变量、调用一些系统功能程序，与系统有较密切的关系，而其它高级语言不具备这种特性。
- 一般来讲 shell 易写易调试，维护成本低，灵活性较强，但执行速度也许不太理想。高级语言通常有较快的执行速度。
- shell 是命令级语言，已解决的问题不必再写程序，shell 允许用户将已经解决

的问题的程序联合串接在一起，这通常缩短了程序的开发时间，在上机实习时也显得更有效益。

- **Unix**系统的高级语言程序都在**Shell**控制下工作。

5. **Unix** 操作系统国际标准化工作已经在顺利地地进行，**Unix** 标准化小组已于 1986 年 4 月推出 **POSIX (Portable Operating System Based on Unix)** 标准后 **IEEE P1003** 委员会又在实现用户界面 **Shell** 及其它工具的标准化，但目前，在 **Unix/Xenix** 系统上运行的 **Shell** 版本仍然很多，大致有如下几种：

① **B Shell**：它是由贝尔（**Bell**）实验室的鲍恩（**S.R. Bourne**）编写的，取编写者姓名的英文首字母 **B**命名。这是在所有 **Unix/Xenix** 系统中都有的一种 **Shell**，因此而又称为标准 **Shell** 通常它在 **/ bin** 目录下名为 **Sh**，即 **/ bin / sh**。该 **Shell** 程序结构紧凑，仅需要较少的系统资源，且执行速度较其它 **Shell** 版本快，处理执行功能也较强。

② **C Shell**：它是由加利福尼亚大学伯克利分校的比尔·乔伊（**Bill Joy**）编写的，是一种具有 **C** 语言程序设计风格的命令解释程序和编程语言，由于和 **C** 语言有许多相似之处而得名。这也是一种在大多数 **Unix/Xenix** 系统中都有的一种 **Shell**，通常在 **/ bin** 目录下名为 **cs**。**cs** 几乎具备了所有 **B Shell** 功能，还具有命令历史功能（命令记忆和恢复）、作业控制、命令别名等功能，并以 **C** 语言语法风格编程，推出 **C Shell** 的主要目的是加强 **Unix** 的交互功能，使用户会话环境更完善，但 **C Shell** 在执行速度和处理功能上不如 **B Shell**，因此一个合适的使用经验是以 **C Shell** 作为注册 **Shell**，而用 **B Shell** 编写问题处理程序。但要注意的是 **cs** 不与 **B shell** 或 **ksh** 兼容。

③ **K Shell** 即由 **David Korn** 编写的 **Shell**，它与 **B Shell** 向上兼容，即在 **B Shell** 下运行的任何命令文件在 **K Shell** 下也可进行。**Ksh** 与 **B shell** 的主要区别之一在于它提供了内部数值处理功能。**Ksh** 吸收了 **C Shell** 的命令历史、作业控制、命令别名等功能，使会话型命令的使用更加方便灵活。**Ksh** 是实现了 **B Shell** 的程序设计环境功能和 **C Shell** 的用户对话环境功能的综合产物，还增加了其它 **Shell** 所没有的加密（保密）保护功能。

④ **tcsh (Temex-Like C-Shell)**：**tcsh** 和 **newcsh** 是在 **C Shell** 基础上扩展而成的两种 **Shell**，其特点是改善了对话环境，其扩展部分的用户界面来源于 **DEC 20** 系列小型机操作系统 **TEMEX**，实现了与后者基本相同的功能。**tcsh** 的最大特点是更直观地实现了命令历史的再使用，它通过使用类似 **Emacs** 的简单行编辑器，用简单的键操作很快可再现使用过的命令，经过适当的编辑可做为新的命令再执行。**tcsh** 还有一个特点是不把重复的命令保留在命令历史表中。在实际应用中，**tcsh** 比 **newcsh** 好用，但两者都出于 **cs** 而胜于 **cs**。

⑤ **V Shell**：这是由 **Microsort** 公司为 **Xenix** 系统新增的一种 **Shell**，在一般 **Unix** 系统中没有使用，**V** 取自单词 **Visual** 因此又被称为“直观 **Shell**”。这是一种把 **Shell** 的部分功能以屏幕菜单驱动方式提供给用户的 **Shell**，它驻留在目录 **/ usr / bin** 下，名为 **vsh**。

上述五种非 **B Shell** 以及其它版本的 **Shell** 都是标准 **Shell** 的有限修改版本，统称为 **rsh**。每一种 **Shell** 都为用户提供了一种界面，完成用户与操作系统的接口工作，每一种 **Shell** 都具备各自的特点，有不同的命令语法规则，提供了不同的功能，按照用户的不同需要和目的用在不同的场合。泛泛比较不同 **Shell** 版本之间的优劣是意义不大的。

系统在用户注册过程的末尾，根据在系统注册或用户注册时的 **Shell** 版本设置

(`/etc/passwd` 文件 `SHELL` 变量) 选择启动一种 `Shell`，以设置用户使用环境，见第三章。

本书主要讲述 `B Shell`，即标准 `Shell`，所有可执行程序例均在 `Unix System V.3.1` 的 `ATT 3B15` 超级小型机上验证通过。

### 三、`shell` 命令的执行过程

1. `shell` 从一个文件读取命令行或语句。如果 `shell` 是交互式的，它首先提示用户键入命令并且从定义在目录 `/dev` 下的终端特别文件上接收命令 (`/dev` 目录与各终端相关联)。如果 `shell` 是用来解释 `Unix` 系统文件的命令行的，将没有提示，而是从文件读取命令。

如果设置了 `mail` 变量，`shell` 还将在给出提示之前通知接收邮件的用户 (只对交互式 `shell` 而言)。

2. `shell` 读取一条命令行之后，它将对命令行执行一系列置换和解释，包括下面几个步骤

- 读命令行 (从终端或文件)，根据空格 `SP` 和制表符 `HT` 对命令行进行初步语法分析  
  变量置换 (用变量的值置换)
- 命令置换 (用指定的命令输出值置换)
- `I/O` 重定向
- 变量赋值
- 字解释——根据列在内部字段分隔符 `IFS` 变量中的字符进行再次命令行语法分析
- 文件名扩展生成 (即元字符、通配符等的扩展)
- 搜索命令文件 (命令定位)

注：元字符是指在某种场合下具有特殊含义的键盘字符。如果你希望使用元字符而不用其特殊含义，就必须用引号括起来。

3. 执行命令行：`shell` 完成解释之后，通过使用 `Unix` 系统原语 `fork(2)` 和 `exec(2)` 执行命令行。

### 四、选择 `shell` 程序设计

`Shell` 可以通过所提供的各种原语 (特殊命令)、条件控制、循环结构语句以及各种参数变量赋值形式等高级特性的组合，将 `Unix` 系统中所提供的各种命令、可调用的实用程序、应用程序及可执行的用户程序组合在一起，进行程序设计，以完成各种所希的功能。

对于一给定的功能并没有固定的准则来决定是否应该使用 `shell` 程序语言。若程序的执行速度很重要则应该考虑那些更有效的语言，在使用 `shell` 的一些高级特性时，执行速度是受到很大影响的；但对于特定的问题，执行速度也许并不是唯一的考虑准则。

- 当一个问题的解决办法中包含了较多的 `Unix` 系统的标准命令操作，应当考虑用

shell 程序设计, Unix 系统命令包括:在文件内查找、将文件排序、传送文件、建立文件、移动文件等。如果问题能用 Unix 系统中已经建立的基本操作来表示,使用 shell 语言将会构成很强的功能。

- 考虑 shell 语言适用性的另一个角度是问题将要涉及的数据类型。若基本数据是正文行或是文件,则 shell 可能构成一个很好的解决办法。若基本数据是数或字符,也许 shell 不太适用。
- 还有一个准则是程序的开发成本。因为交互式语言具有易写易调试的特点,这对于只用很少次数的程序,用 shell 进行编程的开发成本是很合算的。这既可以获得 shell 程序易于开发的优点,又能容忍其较慢执行速度的缺陷,而用其它诸如 C 或 FORTRAN 等高级语言可能花费较多。
- 建议将 shell 语言与 C 语言结合起来解决一些问题,你会惊喜地发现,这是一个非常有效的手段。

## 五、Shell 命令的使用方式

需要说明的是,在 unix 系统中,所谓命令是指由用户或系统定义的完成一个专门功能的(比如 login 命令)可执行文件; shell 命令就是一个以回车作为结束的命令行;因此,所有 Unix 系统命令、可执行的实用程序、应用程序及用户程序,从 Shell 角度来看,都可称之为 Shell 命令。Shell 命令语言本身还具有一些原功能命令和特殊语句结构。在 unix 系统中所有命令都是作为进程来执行的。

1.简单执行方式——一个简单的命令就是以空格或制表符隔开的一串(一个或多个)字。最简单的命令就是单个的字。

格式: `$ command -option argument ..... <CR>`

命令行中第一个字是命令名,接着是任选项和自变量。任选项以“+”或“-”为前缀。任选项可以是分离的或组合在一起的。自变量用于向命令(程序)传递附加控制信息。整个命令行,只有命令名是不能省略的。

例: `$ pr -5tw80 file.1 <CR>` 等价于 `$ pr -5 -t -w80 file.1 <CR>`

都是不带头标(任选项 -t)地将文件 file.1 以每行 5 列 80 个字符的形式显示出来。

2顺序执行方式:

顺序执行是指在一行里用分号(;)分离的一系列命令的依次执行。(在同一行中命令从左到右依次执行)。

格式:

`$ command1 -option1 arg1; command2 -option2 arg2; - <CR>`

在第一个命令完成之后才执行第二个命令。当一系列无关的命令需要在同一时刻执行时,顺序执行便极为有用。如果使用顺序执行,就不必等待一个命令完成之后才请求键入第二个命令。

```
例： $ who ; pwd ; date <CR>
      root    console   Sep 1 09:50
      zhao    tty012    Sep 1 08:50
      /usr / zhao
      Fri Sep 1 09:51:40 BJT 1989
```

\$

顺序执行后各命令的输出依次连接在一起显示。

### 3.命令组方式：

命令组用来组织一组命令使其既可以重新定向整个组的输出又可以作为后台的顺序命令组执行。命令组通常要使用圆括号 )。

```
例： $ (cat encydropedia > mm.ency;cat dictionary > mm.dict) < CR >
```

在此例中，两个 `cat` 请求依次执行，而整个组都处于后台进程中执行。

命令组既可用圆括号，也可用花括号；但是两者的使用格式不一样，在用圆括号时命令组最后一个命令的分号可有可无，但用花括号时则必不可少，而且在 { 后面及 } 的前面必须空一格；但是，当命令使用分行方式而不使用分号时两者的格式则完全一样：

```
例： $ { pwd;who; } > zh.f < CR >
      $ { < CR >
      > pwd < CR >
      > who < CR >
      > } > zh.f < Ctrl-d >
```

\$

还有一个重要的区别，`cd` 命令在圆括号内时只在命令组执行期间临时改变目录，而在花括号中执行 `cd`，则修改了当前工作目录：

```
例： $ pwd < CR >
      /usr / zhao
      $ ( cd /usr / chen;pwd) < CR >
      /usr / chen
      $
      $ pwd < CR >
      /usr / zhao
```

\$

即：在命令组执行前和后的目录一样，`cd` 只是在组执行期间被修改。但在花括号中则不一样：

```
例： $ pwd < CR >
      /usr / zhao
      $ { cd /usr / chen;pwd; } < CR >
      /usr / chen
```

```
$  
$ pwd <CR>  
/usr/chen  
$
```

#### 4.管道线

管道线是产生管道操作的命令线，是由管道操作符分隔的一个命令序列。管道使得左边的一个命令的输出作为其右边一个命令的输入，这是 **Unix** 操作系统特有的极有用的特性之一，在 **shell** 编程中极为有用。管道操作符号是|或。详细讨论见第十七章。

```
例： $ who cut -c1-8 <CR>  
root  
zhao  
chen  
user1  
$
```

首先命令 **who** 产生一个输出，此输出通过管道作为命令 **cut** 的输入，当 **cut** 完成功能处理后即得到例中的结果。由于管道具有重定向功能，只有管道线上最后一个命令 **cut** 的输出才在标准输出上显示，**who** 的输出不显示出。

5.所有上述的四种执行使用方式，又可分为前台执行使用和后台执行使用两种方式。

```
例： $ (date ; pr -t zhao.c) |lp & <CR>  
2345  
$
```

创建后台执行的命令符 & 使整个命令行都在后台执行。见第二章讨论。

## 第二章 shell 进 程

本章是极为重要的，如果你对 Unix 系统缺乏了解，在阅读时你可要认真点儿。在 unix 系统中所有命令都是作为进程来执行的。什么是进程？如果你不能很好地理解进程这一概念，就很难弄清楚 shell 执行过程是怎么回事。本章首先讲述了 shell 进程的概念、功能，然后对 shell 进程的创建、调用、前后台进程、父子进程、命令或程序执行过程中各进程之间的关系和相互作用等问题分别作了描述。显然，本章节是你必须阅读的，不管你心情如何。

shell 是一种实用程序，是你的 Unix 系统不可缺少的组成部分，它从一个文件读取命令、解释命令然后调用 Unix 系统原语执行这些命令。当 shell 被调用时，它便成为一个进程（一个正在运行的程序）。当你注册时，就启动了你的系统中的 shell，通常称这为注册 shell。当你键入 Ctrl-d 注销时，注册 shell 便终止了。

### 一、shell 进程

1. 当 shell 被注册过程启动时，它便成为一个进程，通常称它为注册 Shell 进程。一个进程是一个进程映象的执行。一个进程的映象包括程序、程序环境（变量设置及其变量值）、一组寄存器、一个文件描述字表 FDT（包含怎样打开文件的一些信息的表）、当前目录和根目录名以及其它信息。进程在系统进程表中占一项。

区分程序和进程是很重要的。例如 sort 是一个程序，一个排序程序。每当你运行一次 sort，便产生一个新的进程。假如同一时间有几次运行同一程序，那么每次都产生不同的进程，且有不同的进程标识号 PID。进程的标识号是系统赋值的且是唯一的。

shell 进程用来解释和执行命令行，在注册期间可能在同一时刻有多个 shell 进程在运行。一个 shell 进程解释单级别的命令（来自单一文件的命令）。例如：一个 shell 进程可能解释来自终端键入的命令，而另一个 shell 进程则负责解释被交互式 shell 唤醒的一个 shell 程序中的命令行。

例：`$ update file * <CR>`

功能：自变量 file \* 将被 shell 命令 update 修改。

本例中注册 Shell 进程（父进程）解释来自终端的命令行 `update file *`，当字 `file *` 被解释时，注册 shell 进程执行文件名扩展。另一个 shell 进程解释 shell 程序（命令）`update` 中的命令，此进程便是所谓的子进程。

2. 创建新的进程：当需要时通过 Unix 系统原语 `fork` 便可创建一个新进程，建立的进程包括环境、当前目录和根目录的信息及文件描述字表 FDT，新进程使用它们自己的所有这些信息的副本。进程标识号 PID 在 1~30000 之间是唯一的，并且系统在创建每一个新进程时都赋予一个 PID。

创建一个独立的新的进程是必需的：通过使用独立进程，操作系统提供了在后台运行命令的功能。当运行一个已编译的程序时，因为用来运行编译程序的 Unix 系统原语将覆盖（取代）它的调用进程，也需要使用独立的进程。

创建新的进程（子进程）的唯一方法是调用 `fork` 原语。新进程的内存映象是 `fork` 的调用者的一个副本，只有一点区别，就是老的（父）进程中的返回值包含有新的（子）进程的进程标识号 `PID`，而在子进程中返回值为 `0`。`wait(2)`要利用这个 `PID`。在 `fork` 之前打开的文件要被共享，并且有一个公共的读/写指针，尤其是，这正是标准输入和输出文件的传递和管道安装的途径。

进程的创建方式有前台进程和后台进程两种，创建后台进程时在命令行后加 `&` 符。

创建新进程时也可能失败，失败时返回值为 `1`，这可能是因为：

- 系统强制定义的进程数目达到限定值，这个限定值是整个 `Unix` 系统限定的全部进程数目的最大值（即：系统进程表满）。
- 系统强制定义的用户进程数目达到限定值，这是系统为每一个用户限定的值。
- 内部进程通讯出错。

这些限定值可通过下面命令显示：`$ / etc / sysdef grep proc <CR>`其中 `procs` 是系统限制，还有一个 `maxproc`是用户限制。在一个实验系统里，限定值为：

```
$ / etc / sysdef grep proc <CR >
procs      150
maxproc    25
$
```

3.执行读入命令行的 `shell` 进程称作父进程，执行出命令行结果的进程便是所谓的子进程，每一个新建的进程（通过 `fork`）都从创建它的父进程那儿继承一些信息，这通常包括：

- 程序映象
- 当前目录和根目录信息
- 环境（一系列变量及其值）
- 信号处理信息
- `Umask` 设置（隐含文件权限限制）
- `Ulimit` 设置（最大文件尺寸）
- 文件描述字表 `FDT`

4.当运行命令：

```
$ pr zhao.c |lp & <CR >
1234
$
```

时，由于符号 `&` 作用到整个管道线，因此所有的进程都立即起动，但终端上只显示一个进程标识号，这个 `PID` 是一系列进程中最后一个进程的 `PID`。本例中 `1234`便是 `lp` 的进程标识号。

5.如果忘记了进程标识号，用 `ps` 命令可以查询当前正在运行的进程 `PID`。`kill` 命令可

以通过该 PID 终止进程的执行。

- 6. 进程和文件系统有类似的层次结构。每个进程有其父进程，还可以有其子进程、孙进程或者说后代进程。

例如：注册 shell 是由专门的注册进程启动的，这一进程主要管理系统与终端间的连结。运行命令的进程是注册 shell 进程的直接子进程。

如果在这些命令的进程中再运行别的程序，如在编辑命令 vi 中再使用命令，vi 就生成了一个自己的子进程，而这一子进程便是注册 shell 的孙进程。

- 通常，shell 执行一条命令需要三个进程：

执行命令行的读入、解释、判断和信息提示的进程。

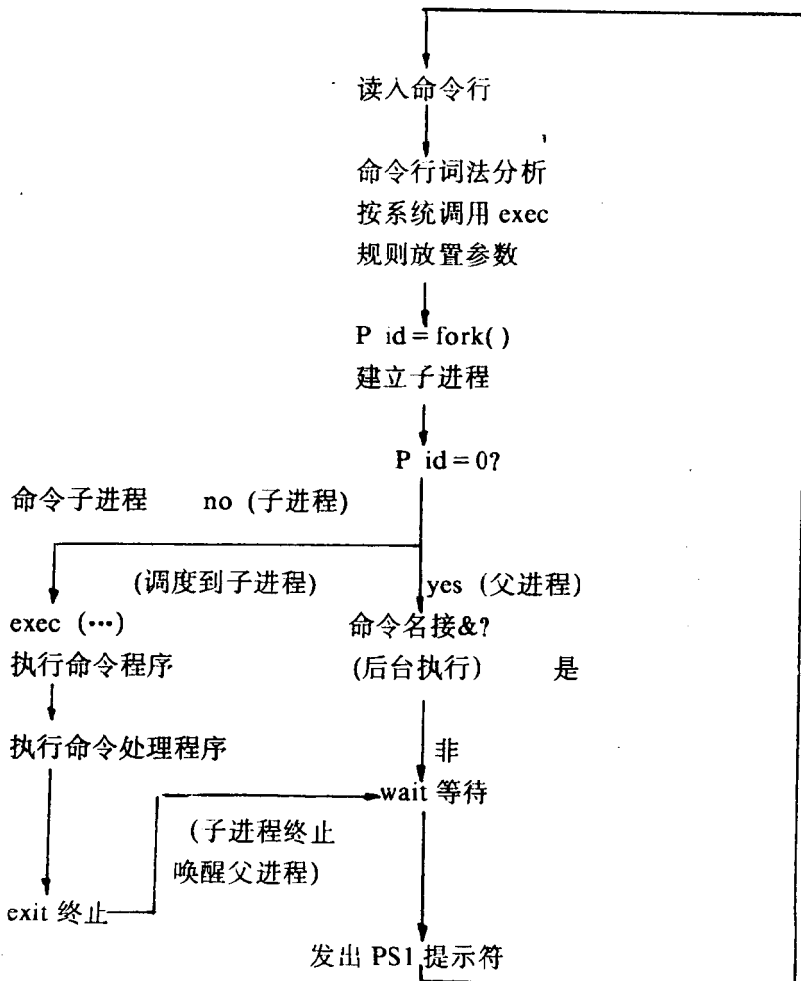
执行命令行本身功能的进程。

命令行结构中调用其它功能时创建的进程：比如编辑命令 vi 再使用命令，管道线中的命令等。

- 很显然，shell 的执行速度在很大程度上受创建进程的影响。

下面是一条 Shell 命令执行过程中进程之间的调度关系图示：

Shell 命令树节点命令的 Shell 进程

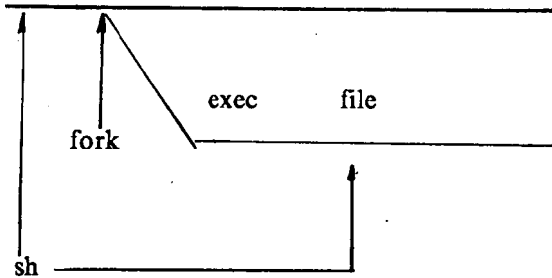


为了执行一个 Shell 命令树中节点命令，除 Shell 原语等极少数情况外，一般都要先调用 fork 创建子进程，子进程继承父进程的环境和特性并建立自己的特性；子进程调用 exec，用命令对应的程序替换进程影象并执行；通常一旦子进程被建立，父进程一般要等待子进程终止（wait）；子进程执行完毕（终止）后唤醒父进程，然后执行 Shell 命令树其它命令节点。父进程或者本身也消亡，或者进行另一命令行的读入。可以说，Shell 命令解释程序的工作过程就是父子进程的创建、执行、终止和等待的同步协调过程。进程的创建数目和效率在很大程度上影响着 Shell 的执行速度。

## 二、编译后程序及 shell 过程的执行

1. 当编译后的程序被执行时，shell 解释执行命令行，创建一个新的进程（fork），然后使这一进程去执行这个编译后的程序。当其子进程执行编译文件时，它将毁坏（覆盖）此进程。

例：       \$ file <CR>



用来执行一个编译后程序的系统原语命名为 exec，它将目前正在运行的进程转换成一个新的进程。在本例中，交互式注册 shell 解释键入的命令是什么，调用 fork 原语创建一个新的 shell 进程来判断是编译后的程序还是 shell 程序。如果是编译后程序（不是 shell 程序），该 shell 进程将牺牲自己来运行编译程序。因此，为了执行一个编译后程序，shell 将：

- 解释命令行
- 建立子进程
- 子进程执行编译后程序并覆盖子进程自身

### 2. 执行 shell 过程

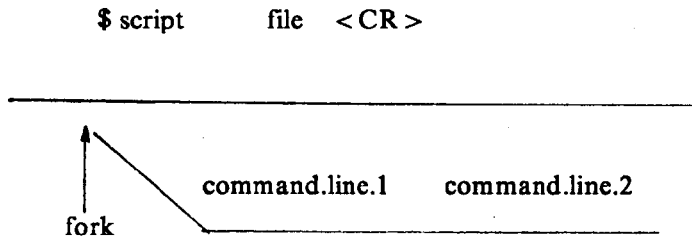
当执行一个 shell 过程时，交互式（注册）shell 解释命令行，创建新的进程（fork），然后由此进程解释此 shell 过程。在下面例子中 shell 程序 script 由注册 shell 激活，程序的内容如下：

```
$ cat script <CR>
command.line.1
```

command.line.2

\$

下图显示了为运行 shell 程序 script，系统将需要做些什么？



本例中，系统首先要调用 `fork` 创建一个进程，它必须执行两个命令行的解释，对于每一行如果它们不是原语 shell 命令，则建立一个独立的子进程来执行 `command.line.1` 和 `command.line.2` 命令。原语命令是独特的，不需要产生独立进程来执行原语命令。因此，为了运行一个 shell 过程，shell 需要：

- 解释命令行
- 创建一个子进程
- 使此子进程解释 shell 过程

如果一个进程作为前台（隐含的）建立，则父进程要等待子进程完成后才继续，这是通过父进程发出系统原语 `wait(2)` 完成的。总之，当一个前台命令行从终端键入时，交互式 shell 将要：

执行键入命令行的解释  
创建一个新进程，使它执行命令或解释 shell 过程  
等待子进程结束  
提示或等待下一个命令

三、执行 shell 程序（命令文件）的方法：

Shell 可以进行程序设计，当程序被 `Vi`、`cd` 等工具编辑存入系统后执行 shell 程序（或 shell 命令文件、shell 过程）有三种方法：

- 重定向法      `$ sh <file-name <CR>`
- 自变量法      `$ sh [-vxenu] file-name <CR>`
- 增权法        `$ chmod [agou]+x file-name <CR>`  
                  `$ file-name <CR>`

重定向法是一种间接执行的方法，它不要求文件具有执行许可权，只要有读许可便可以执行。自变量方式是调用 **sh** 实用程序来执行的，具有多种调试功能，当你刚建立一个 **shell** 程序时，使用 **sh** 的调试任选项可以发现、检测和校正错误，这显然具有明显的优越性。当程序准确无误后，如果程序要经常执行，使用增权法便具有很大的方便性，直接引用文件名便可执行。关于程序的调试和执行举例见第六章。

#### 四、执行 **shell** 命令（文件）的方式：

执行 **shell** 程序（或 **shell** 命令、命令文件、过程）有二种方式，即前台执行方式和后台执行方式。除非明确指定为后台执行，则命令都在前台执行。前台命令必须执行完毕后才能进行下一个命令的执行，后台命令执行则不一定。在前台执行时创建前台进程，在后台执行时创建后台进程，后台进程不需要监督。

1.前台进程：相对于后台进程而言的一种交互运行的进程。如果没有说明，通常所讲的进程便是指系统隐含约定的前台进程。在正常环境下，只能有一个活动的前台进程、或许还有一些不在活动的（睡眠状态的）前台进程。

2.后台进程：在一定意义上说，这是一种运行时无需监视管理的进程。当执行一个后台命令时，**unix** 系统创建一个子进程，给请求的命令赋予一个进程 **PID**，一旦进程赋予 **PID** 便放弃 **shell** 控制权，返回 **PID** 及 **shell** 提示符，因此，当后台进程运行时其它程序仍可以开始和对话。当 **Unix** 系统引导时有一些后台进程被启动以便执行系统管理功能，另一些后台进程由用户以交互方式启动，以便执行该用户的工作。

有时后台进程被称作异步进程，并独立于其它进程。后台进程忽略通常用于终止 **shell** 命令的中断（通过 **break**），但是不忽略断线和退出信号。为了终止后台进程一般要用 **kill** 命令。后台进程有一个由文件 **/dev/null** 定向的标准输入。因此，一旦希望后台进程从标准输入读入，则立即可接收到 **EOF (end-of-file)**。通常，后台运行的命令输出都被重新定向。如果没有，此进程写一个信息给标准输出和标准错误输出，并在终端上显示。因为后台进程忽略中断，**break** 将不会影响输出。

#### 后台进程

也称异步进程

独立运行于其它进程

有一个标准输入从 **/dev/null** 提供

将忽略中断 **break**

不忽略断线和退出信号

通常有输入和输出重定向

3.创建后台进程：有时要运行一个时间较长的程序，如果该程序的运行不需要用终端再输入信息，那么完全可以不用监视地运行它，此时后台运行是合适的。

后台进程通过在命令行后面加上符号 **&** 创建，单个用户可以在同一时间运行多个后台进程。在 **Unix** 操作系统下运行的后台进程优先级并不比前台进程的低，后台进程

和前台进程通常是同时运行。

当一个后台进程产生时，在标准错误信息输出上会显示一个进程标识号。当使用下面的后台进程管理工具时，需要这个标识号：

- **ps** 显示进程状态
- **wait** 等待后台进程完成
- **kill** 终止后台进程

引用后台进程的另一个有用工具是 **nohup(1)**。

**4.ps**命令：用来显示正在运行的进程的有关信息。如果不带任何任选项，则显示如：

```
$ ps <CR>
```

| PID  | TTY | TIME | command |
|------|-----|------|---------|
| 291  | 00  | 0:06 | sh      |
| 8805 | 00  | 0:01 | ps      |
| :    | :   | :    | :       |
| :    | :   | :    | :       |

**PID** 是进程标识号  
**TTY** 是终端特别文件号  
**TIME** 是进程执行时间的总计数  
**command** 是命令名

如果使用 **-f** 任选项，**ps** 命令将输出全部状态列表，如下例：

```
$ ps -f
```

| UID   | PID  | PPID | C  | STIME    | TTY | TIME | command |
|-------|------|------|----|----------|-----|------|---------|
| huteh | 291  | 1    | 0  | 07:22:51 | 00  | 0:06 | -sh     |
| huteh | 8829 | 8828 | 55 | 11:28:53 | 00  | 0:01 | ps-f    |
| :     | :    | :    | :  | :        | :   | :    | :       |
| :     | :    | :    | :  | :        | :   | :    | :       |

**UID** 是正在运行的进程的用户注册名  
**PID** 是进程标识号  
**PPID** 是父进程标识号  
**C** 是进程序列信息  
**STIME** 是进程启动的时间

一条 **ps** 命令涉及到两个进程，一是注册时启动的 **shell** 进程，在注册过程的末尾就自动激活了，二是在你键入 **ps** 命令时被激活的 **ps** 进程。

动激活了，二是在你键入 `ps` 命令时被激活的 `ps` 进程。

### 5.wait 语句

`wait` 语句用来使交互式 `shell` 等待 & 产生的后台进程的结束（也用于父进程对子进程的等待）并报告异常终止。`wait` 由注册 `shell` 执行，不需要产生新的进程。`wait` 命令需要你提供引起交互式 `shell` 等待的进程的标识号。一个进程可能只等待它的子进程，而不是它的所有后代进程（孙进程）。

如果 `wait` 语句没有任何变量，则等待最后一个子进程的完成（`die`），在这种情况下，交互式 `shell` 再继续。

如果等待管道线，管道线里最后一个执行的命令则成为 `wait` 的目标。

当一个进程需要进入后台运行、进行一些处理然后等待进程的完成结束时，`wait` 是有用的。

### 6.kill 命令

通常，`kill` 用来终止一个进程。如果是后台进程就需要提供后台进程的标识号，并立即终止此进程。当 `kill` 引用时，隐含地发出一个信号 `15` 给进程以进行终止。进程也可能忽视此信号，如果这样，`kill` 就需带 `-9` 任选项以终止此进程，这就是所谓“无条件 `kill`”（`uncatchable kill`）。

如果进程标识号是 `0`，则 `kill` 将给目前 `shell` 的所有子进程发出终止信号。除非是超级用户，否则被 `kill` 的进程必须属于用户自己。超级用户可用 `kill` 来关闭系统并转换成单用户状态。如果给出用 `_` 作前缀的信号数值作为 `kill` 命令的第一个自变量，就发送该信号给用 `PID` 指定的一个或多个进程，而不一定终止进程。使用格式如下：

**kill [-Signo] PID...**

关于信号说明见第十八章。

如果下面一种或几种情况为真则 `kill` 失败，并且没有信号发送，命令返回状态值为 `_1`：

- .指定的信号是不可用信号。
- .信号是 `SIGKILL` 且 `PID` 是 `1`（进程 `1: proc1`）
- .指定的进程 `PID` 不存在。
- .发送信号的进程的用户标识号不是超级用户，并且它的实际或有效用户标识也不能匹配接收信号的进程的实际或有效用户标识号。

### 7.nohup 命令

通常在系统注销时，已经在后台工作的程序将会收到一个“挂起”（`HANGUP`）信号，多数程序一旦收到此信号便退出。`nohup` 命令正是用于禁止退出的，即当你从系统中注销时，冠有 `nohup` 命令的后台进程不会死亡。

如果将 `nohup` 前置在任何一后台进程命令行前，它将使进程照常执行而同时忽视 `HANGUP` 信号及 `logoff` 信号（还有 `INTERRUPT` 及 `QUIT`）。

如果输出没有重定向，则将标准输出和标准错误输出写入当前目录下一个名为 `nohup.out` 的文件。如果当前目录限制建立文件的权限，或者此文件存在但指定了权限，

则此文件建立在本用户注册目录下，即将输出重定向到 **\$HOME/nohup.out** 文件。

如果你的进程需占用较多的处理机资源，而其它用户与你共享这些资源，在这种情况下，用低优先级就比较合乎情理：

```
$ nice [-increment] longtime-program &
```

其中 **-increment** 是优先级数（缺省为 10）。

其实，**nohup** 自动地调用了 **nice** 功能使命令的优先级增加（通常增加 5）。一般没有再停止 **nohup** 命令的办法。

```
例 1:          $ nohup sort long.file 2 > errors > out <CR>
```

在后台运行 **mm** 命令，忽略 **HANGUP**、**INTERRUPT** 及 **QUIT** 信号。如果标准输出或标准错误输出没有重定向，则将输出增加到名 **nohup.out** 的文件之后；如果此文件不存在，它就被建在当前目录下（如果用户有权在当前目录下建立文件）。

例 2

```
$ nohup sort long.file <CR>  
sending output to nohup.out  
$
```

因为标准输出没有重定向，**nohup** 则将输出送到文件 **nohup.out**。

关于 **ps**、**kill**、**wait** 和 **nohup**、**nice** 的更详尽说明请参见有关 **Unix** 系统资料。