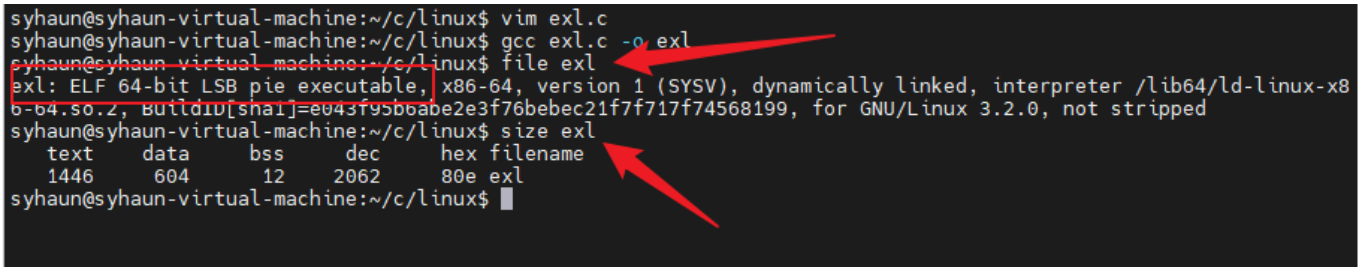
linux 可执行程序的存储结构与进程结构

linux可执行程序的存储结构

linux中可执行文件的格式为ElF,可以使用file或者readelf命令来查看文件的情况.size命令查看可执行文件中各段的大小.

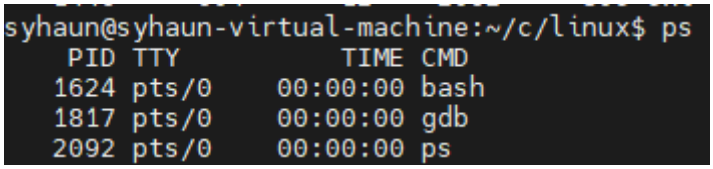


size命令输出结构:dec总 hex 16进制 可执行文件由3段组成

1. 代码段(text):放程序的二进制代码,是CPU唯一能执行的机械指令.通常是只读的,共享的.此外,还规划了局部变量的相关信息. 代码段的机械指令包括操作码和操作对象(或对象地址引用).操作对象是立即数(具体的数值),那么直接包含在代码中,如果局部数据,运行局部数据的函数,系统将局部变量在栈区分配空间,使用数据时,通过引用数据地址的方式读.
2. 全局初始化数据区/静态数据区(data):数据段,存储被初始化的全局变量,已初始化的静态变量(包括全局静态变量和局部静态变量)和常量数据(字符串常量),静态内存分配区.
3. 未初始化数据区(bss):BSS段.保存全局未初始化变量和未初始化静态变量,值未0或空. 在程序文件中未分配空间,仅使用一部分记录大小和属性,加载到进程中分配空间并将数据化进行初始化未0或空.

程序的进程结构

进程源于程序,是程序的一次执行过程,当可执行程序被运行时,操作系统将可执行程序读入内存,依据程序的内容创建一个进程.ps命令查看进程的情况

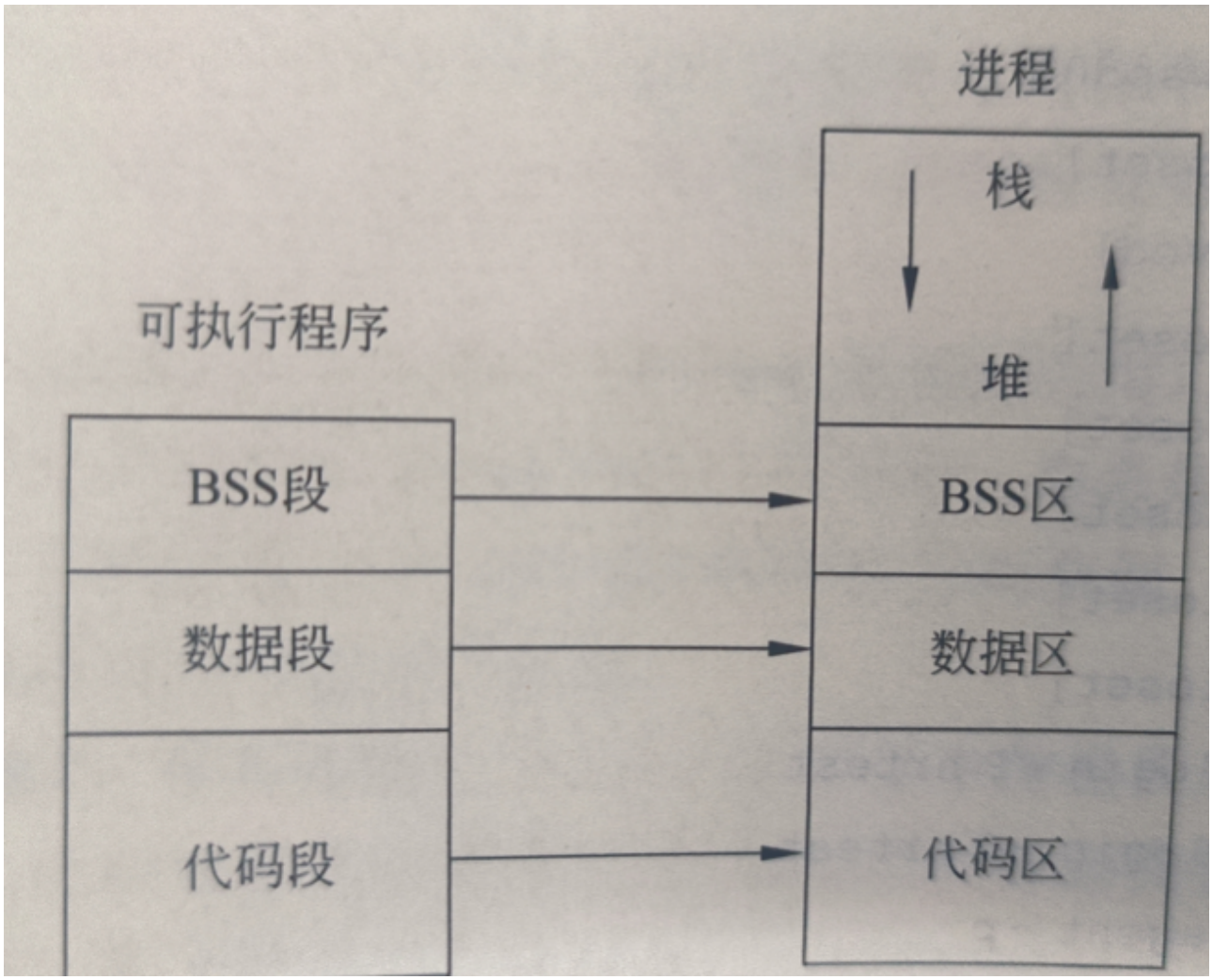


ps命令输出结果:

- 第一列为PID号(非负整数),进程ID号或进程号,每个进程都有一个唯一的进程号标识
- 第二列为USER,创建该进程的用户
- 第三列为COMMAND即与该进程对应的可执行程序名称

进程一般分为5个部分

1. 代码区:存放进程执行的代码,在程序运行前已确定
2. 数据区:存放已初始化全局变量和静态变量,在程序运行前已确定
3. BSS区:存放未初始化的全局变量和静态变量,在程序运行前已确定
4. 堆区(heap):进程运行过程中可被动态分配的内存段,大小不固定,可动态扩张或缩减.程序中使用malloc等函数动态分配的内存属于堆区,不需要时调用free函数将内存释放
5. 栈区(stack):栈区内存由操作系统自动分配,用于存放进程临时创建的局部变量,函数调用时的参数返回值等,函数调用结束后,释放.



进程树

linux中除第一个进程时"手工"建立外,其余进程使用系统调用fork创建,被创建的进程称为子进程,调用fork的进程称为父进程.内核进程使用进程ID号标识每一个进程.一个进程除了有一个PID外,还有一个PPID存储父进程的PID
`pstree`查看进程树

进程的环境和进程属性

进程的环境

进程运行中,常使用创建进程时所设置的命令行参数 Linux中命令大多数由c语言编写 从主函数开始执行,要求main函数接受命令行参数,带参数的主函数原型为

```
int main(int argc,char *argv[],char *envp);
int main(int arfc,char *argc[]);
//argv 命令行参数的个数
//argv 指向每个参数的指针所组成的数组
```

命令行中未设置环境变量,使用默认的环境变量值 `envp`记录了程序运行时的环境变量.环境变量`environ`是一个全局变量,存放在所有的Shell中,登录时就有了相应系统定义的环境变量.linux的环境变量有继承性,子进程继承父进程的环境变量.环境变量是一个指针数组,记录程序运行有关的系统数据(默认路径,Shell类型等)

获取环境变量的值

shell中env,`set'命令查看当前环境变量

c语言中getenv函数

```
char *getenv(const char (name));
```

进程的状态

理论上

1. 运行态
2. 就绪态
3. 阻塞态

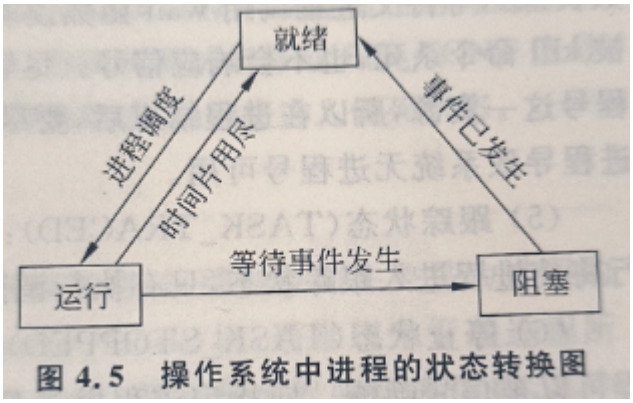


图 4.5 操作系统中进程的状态转换图

实际上

1. 可运行态
2. 可中断的阻塞态
3. 不可中断的阻塞态
4. 僵死状态
5. 跟踪状态
6. 停止状态

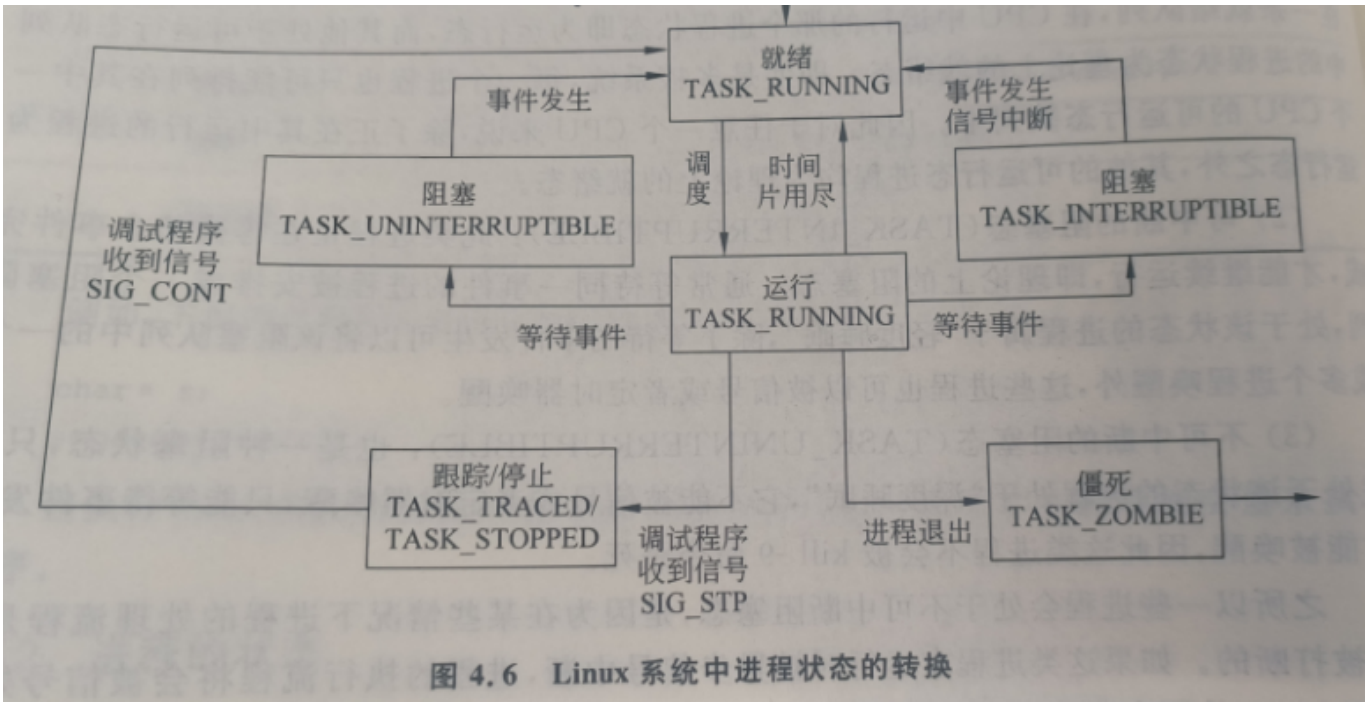


图 4.6 Linux 系统中进程状态的转换

进程的基本属性

进程控制块(PCB块):一个`task_struct`类型的结构体,记录进程的各种信息.

进程号和父进程号

PID号是系统维护的一个非负整数,在创建进程时确定,无法在用户层修改.

使用`getpid()`函数获得PID

```
pid_t getpid();
//成功返回进程号,失败返回-1
```

使用`getppid()`获取父进程号

```
pid_t getppid();
//成功返回父进程号,失败返回-1
```

进程组号

进程组:一个或多个进程的集合,通常和作业相关联,接受来自同一终端的跟各种信息.每个进程都属于一个或多个进程组.每个进程组有个称为组长的进程,组长进程的PID等于进程组号(GID).

`getpgid`获取进程组号,`setpgid`设置进程组号.

```
pid_t getpgid(pid_t pid);
```

```
pid_t setpgid(pid_t pid,pid_t pgid);
//pid:进程PID号,pgid待设置的进程组号
```

将进程号为pid的进程的进程组号设置为pgid.

当pid为0,标识设置当前进程的进程组号

当pgid为0,pid进程的进程组号设置为自己的进程号,即pid进程成为进程组的组长

会话

会话:一个或多个进程组的集合,每个会话有唯一会话首进程,会话号等于会话首进程号.

一般情况,用户登录后所执行的所有程序都属于一个会话,登录的Shell是会话首进程,Shell所使用的终端就是会话的控制终端.

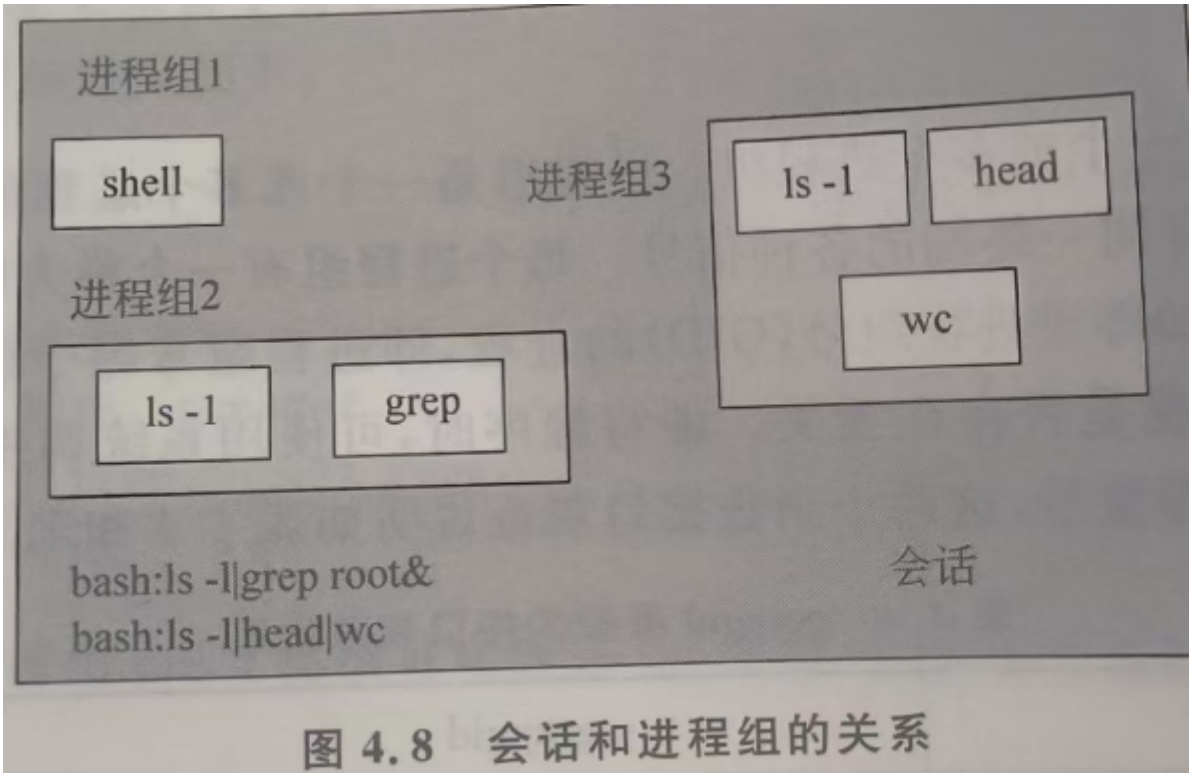


图 4.8 会话和进程组的关系

getsid获取会话ID,setsid创建新会话

```
pid_t getsid(pid_t pid);
//pid为0时表示获取当前进程的会话号
```

```
pid_t setsid(void);
```

调用setsid函数的进程不能是进程组长.

非组长进程调用setsid,创建一个新会话,该进程加入这个新会话并成为这个会话的首进程以及新进程组的组长

控制终端

会话首进程打开一个控制终端,该终端成为此会话的控制终端.会话和控制终端是一一对应的关系.与 控制终端连接的会话首进程称为控制进程.当前与终端交互的进程所在的进程组称作前台进程组,其余进程组称为后台进程组.

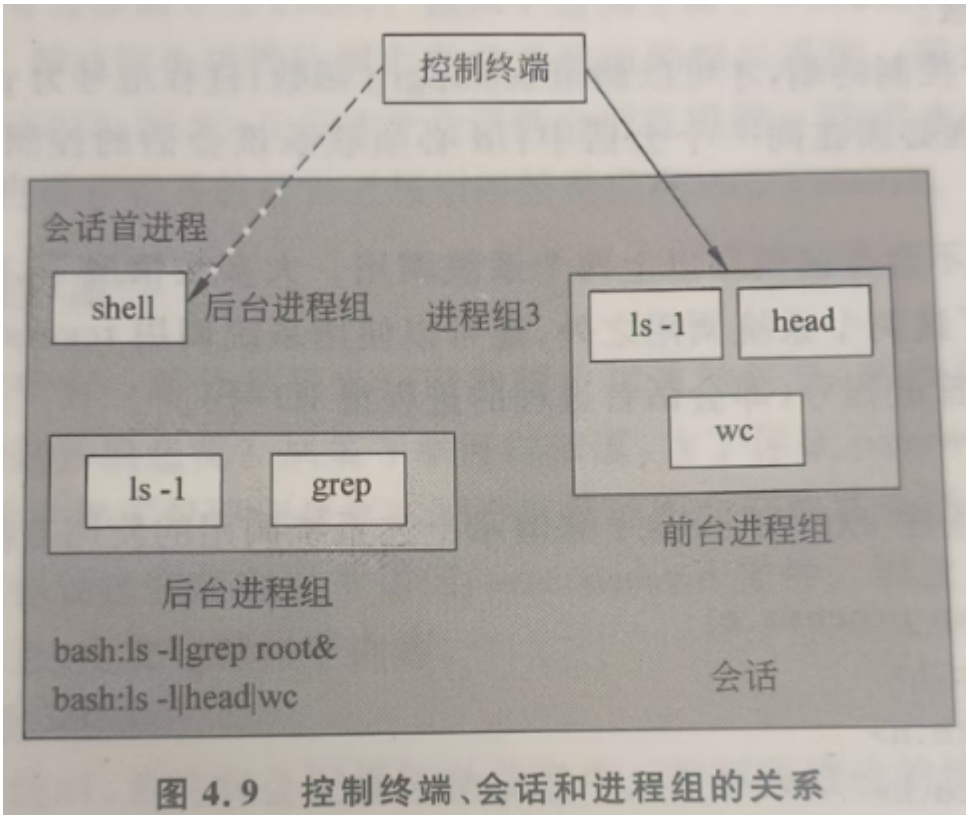


图 4.9 控制终端、会话和进程组的关系

系统调用tcgetpgrp和tcsetpgrp获取或设置与终端连接的前端进程组.

```
pid_t tcgetpgrp(int fd);
//指定控制终端对应的文件描述符
```

```
pid_t tcsetpgrp(int fd,pid_t pgrp);
//指定控制终端对应的文件描述符
//待设置为前台的进程组的组ID号
```

使用tcgetpgrp时,参数fd必须时调用tcgetpgrp进程的控制终端

进程要有一个控制终端,才能调用tcsetpgrp函数,进程组号为pgrp的进程组和调用tcsetpgrp的进程必须在同一会话中,fd必须联系该会话的控制终端,负责调用失败