# 一 、进程基础

## 进程标识符PID (进程号)

pid\_t类型（传统上为16位有符号整型数/现在不确定）

ps常用命令 ( -aux -axf -axm ax -L )

进程号是顺次向下使用（和文件描述符相反）,到最大值后再从头开始。

//获取进程号 和 父进程号

pid\_t getpid(void);

pid\_t getppid(void);

## 进程的产生fork vfock

pid\_t fork(void);

父进程所有数据复制(memcpy)一份成为子进程，两进程，一模一样，执行位置也相同。子进程和父进程里该函数的返回值不同。根据不同的返回值，执行不同的代码分支。

fork后父子进程的不同之处：

* fork()函数的返回值不同
* PID不同
* PPID不同
* 未决信号和文件锁不继承
* 资源利用量清0

init进程（1号进程）是所有进程的祖先进程。

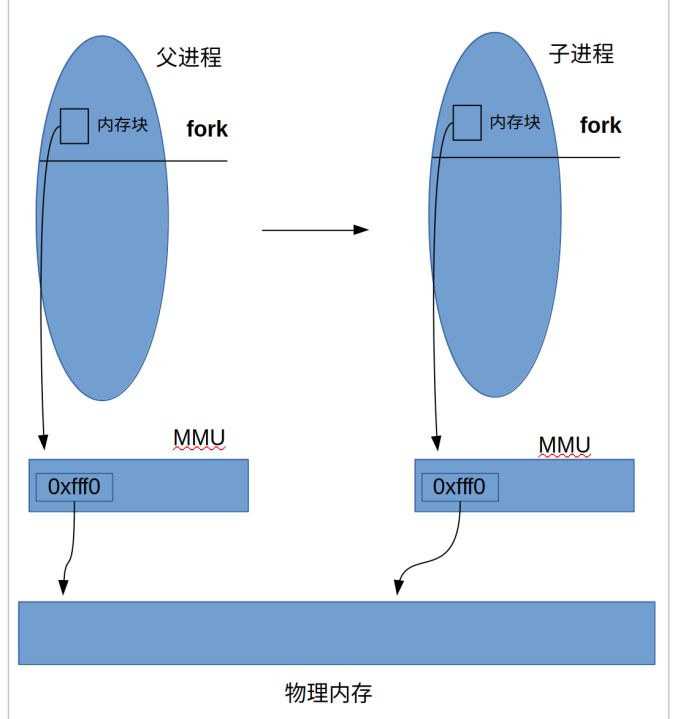
fork后的父子进程 谁先运行与调度策略有关。

fork前刷新缓冲区，缓冲区会继承。fflush(); 刷新后产生的子进程和父进程一样缓冲区为空

## 数据共享问题

pid\_t vfork(void);(不常用)

fork出的子进程和父进程中的同一地址的虚拟内存块，实际在在物理内存的不同位置。如下图：



而vfock出的子进程与父进程使用相同的数据块。

the child shares all memory with its parent, including the stack.

## 写时复制机制

现在的fork加入了写时复制机制。只读数据父子进程共享，谁要写时单独复制一份，对复制的数据进行修改。所以现在vfock很少用。

## 进程的消亡及资源回收

子进程的资源是父进程申请的，一般由父进程释放,父进程等待子进程结束信号去回收资源。

pid\_t wait(int \*wstatus); //死等

pid\_t waitpid(pid\_t pid, int \*wstatus, int options);//有选择的等

The value of pid can be:

< -1 meaning wait for any child process whose process group ID is equal to the absolute value of pid.

-1 meaning wait for any child process.

0 meaning wait for any child process whose process group ID is equal to that of the calling process.

> 0 meaning wait for the child whose process ID is equal to the value of pid.

int waitid(idtype\_t idtype, id\_t id, siginfo\_t \*infop, int options);

/\* This is the glibc and POSIX interface; see

NOTES for information on the raw system call. \*/

pid\_t wait3(int \*wstatus, int options,

struct rusage \*rusage);

pid\_t wait4(pid\_t pid, int \*wstatus, int options,

struct rusage \*rusage);

## 多进程的任务分配

1 分块

2交叉分配 //你一个 我一个 你一个 我一个 类似的方式

3池 （类似消息队列）

## exec函数族

运行一个二进制可执行文件

extern char \*\*environ;//环境变量保证只需要一个程序名就能运行

int execl(const char \*path, const char \*arg, ...

/\* (char \*) NULL \*/); //传参从argv0 开始

int execlp(const char \*file, const char \*arg, ...

/\* (char \*) NULL \*/);

int execle(const char \*path, const char \*arg, ...

/\*, (char \*) NULL, char \* const envp[] \*/);

int execv(const char \*path, char \*const argv[]);

int execvp(const char \*file, char \*const argv[]);

int execvpe(const char \*file, char \*const argv[],

char \*const envp[]);

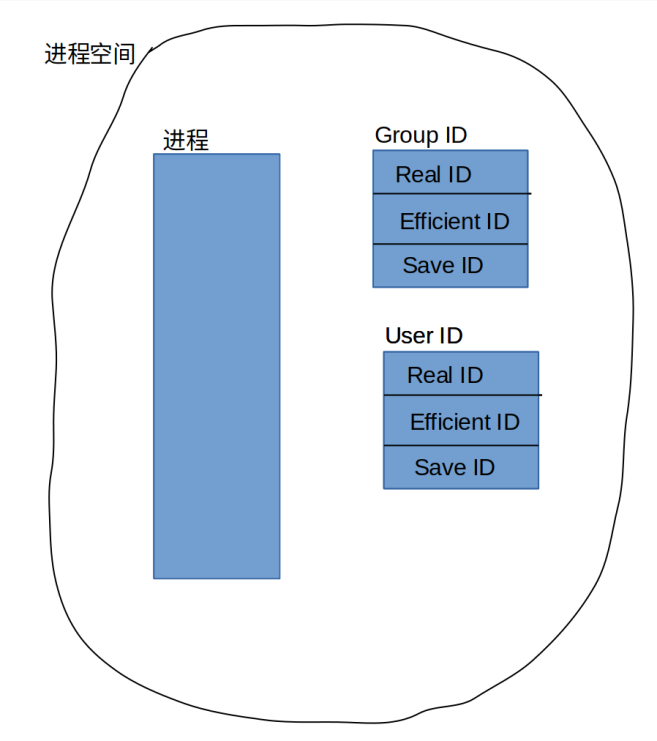
replaces the current process image with a new process image PID不变

exec前注意 缓冲区的刷新

## fork()+exec()+wait()为常用方法，父进程fork(),子进程exec一个文件，父进程等待子进程执行。

## 用户权限和组权限

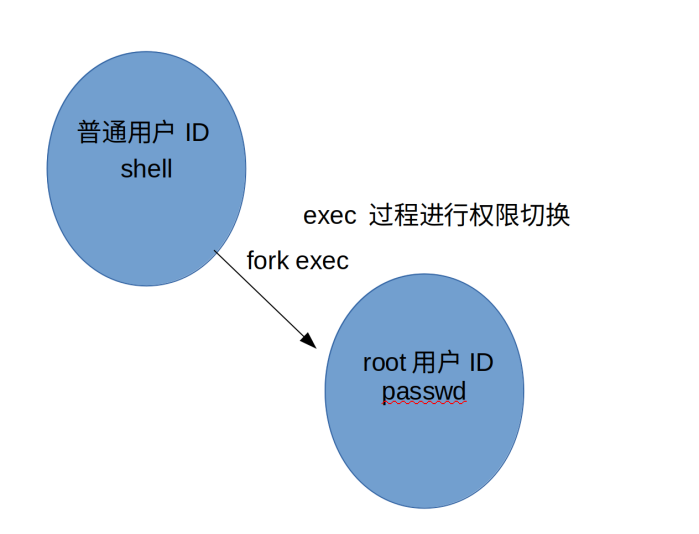
user ID和group ID 都由3个ID组成，rid,eid,sid(有的系统没有)，存在进程空间。



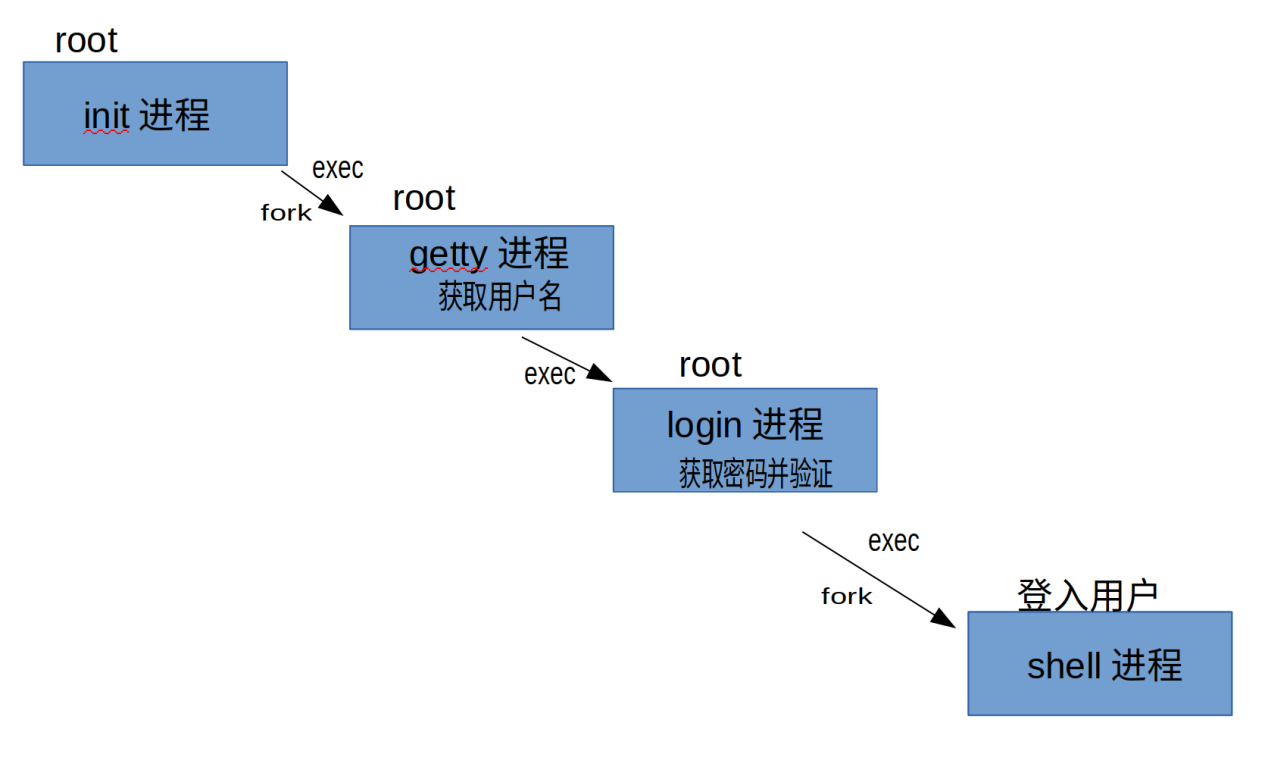
U+S

G+S

进程执行带有 U+S位的可执行文件时，进程的权限将切换为root权限，就是User ID 会变为root的User ID。exec函数发生权限的切换。（权限不用切换回普通用户）



## 登入过程及权限分析



note：

/etc/passwd 文件可以修改每个用户的登入shell。用来 对付棒槌老板，安全要求高的地方。

## UID和GID 函数

uid\_t getuid(void);

uid\_t geteuid(void);

gid\_t getgid(void);

gid\_t getegid(void);

int setuid(uid\_t uid);

int setgid(gid\_t gid);

int setreuid(uid\_t ruid, uid\_t euid);

int setregid(gid\_t rgid, gid\_t egid);

int seteuid(uid\_t euid);

int setegid(gid\_t egid);

note:当要运行的要文件有U+S位，使用setuid更改为root用户运行可执行文件才能成功。

## 解释器文件

脚本文件

其起始行的形式是:

#! pathname [ optional-argument ]

在感叹号和pathname之间的空格是可选的。最常⻅的解释器文件

以下列行开始:

#! /bin/sh

执行此类文件内核exec会只装载#！后制定的程序，然后用这个程序解析整个文件。

也就是#！后可以跟任何可执行文件。

## system()

int system(const char \*command);//执行一个shell命令 对exec调用shell执行命令的封装

## 进程会计

int acct(const char \*filename); //free BSD 方言

## 进程时间

clock\_t times(struct tms \*buf);

//总时间=自己的时间+子进程的时间

struct tms {

clock\_t tms\_utime; /\* user time \*/

clock\_t tms\_stime; /\* system time \*/

clock\_t tms\_cutime; /\* user time of children \*/

clock\_t tms\_cstime; /\* system time of children \*/

};

//滴答时钟

The number of clock ticks per second can be obtained using:

sysconf(\_SC\_CLK\_TCK);

## 守护进程

进程组组长的进程ID 就是该进程组的进程组ID。就是组长进程的进程ID和进程组ID相同。

进程组和会话（session）在进程之间形成了两级的层次：进程组是一组相关进程的集合，会话是一组相关进程组的集合。   
这样说来，一个进程会有如下ID：   
·PID：进程的唯一标识。对于多线程的进程而言，所有线程调用getpid函数会返回相同的值。   
·PGID：进程组ID。每个进程都会有进程组ID，表示该进程所属的进程组。默认情况下新创建的进程会继承父进程的进程组ID。   
·SID：会话ID。每个进程也都有会话ID。默认情况下，新创建的进程会继承父进程的会话ID

使用 ps axj 命令可以查看PID PPID PGID SID

//设置SID 创建守护进程

pid\_t setsid(void);

setsid() creates a new session if the calling process is not a process group leader. The calling process is the leader of the new session (i.e., its session ID is made the same as its process ID). The calling process also becomes the process group leader of a new process group in the session (i.e., its process group ID is made the same as its process ID).

守护进程的特点：

* 脱离控制终端
* PID PGID SID 三者相同
* PPID为1（init进程）

//进程组操作

int setpgid(pid\_t pid, pid\_t pgid);//设置GID

pid\_t getpgid(pid\_t pid);//获取GID

pid\_t getpgrp(void); //获取进程组 /\* POSIX.1 version \*/

pid\_t getpgrp(pid\_t pid); //获取进程组 /\* BSD version \*/

## 系统日志

/var/log 目录下为系统日志

/var/log/messages 为系统主日志文件

## syslogd服务与Rsyslog服务

syslogd服务是专门收集日志的服务，只有syslogd服务有权限写系统日志，程序将log提交到syslogd，由syslogd来写系统日志。

后来Rsyslog取代了syslog。

Rsyslog 是负责收集 syslog 的程序，可以用来取代 syslogd 或 syslog-ng

ubuntu的主日志文件也发生了变化，日志文件存在于/var/log/syslog

而对于配置文件，存在于/etc/rsyslog.conf中(该文件指出了，所有配置文件都在/etc/rsyslog.d目录下)，而不是一般所说的/etc/[syslog](https://so.csdn.net/so/search?q=syslog&spm=1001.2101.3001.7020" \t "/home/ww/文档\\x/_blank).conf中。

配置文件决定了什么级别的日志会写进日志文件。

//建立与系统日志服务（syslogd）的关联

void openlog(const char \*ident, int option, int facility);

const char \*ident, 字段

int option, 特殊要求

int facility 来源

//提交日志

void syslog(int priority, const char \*format, ...);

int priority, 级别

const char \*format格式类似printf

//断开与系统日志服务（syslogd）的关联

void closelog(void);