# 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。

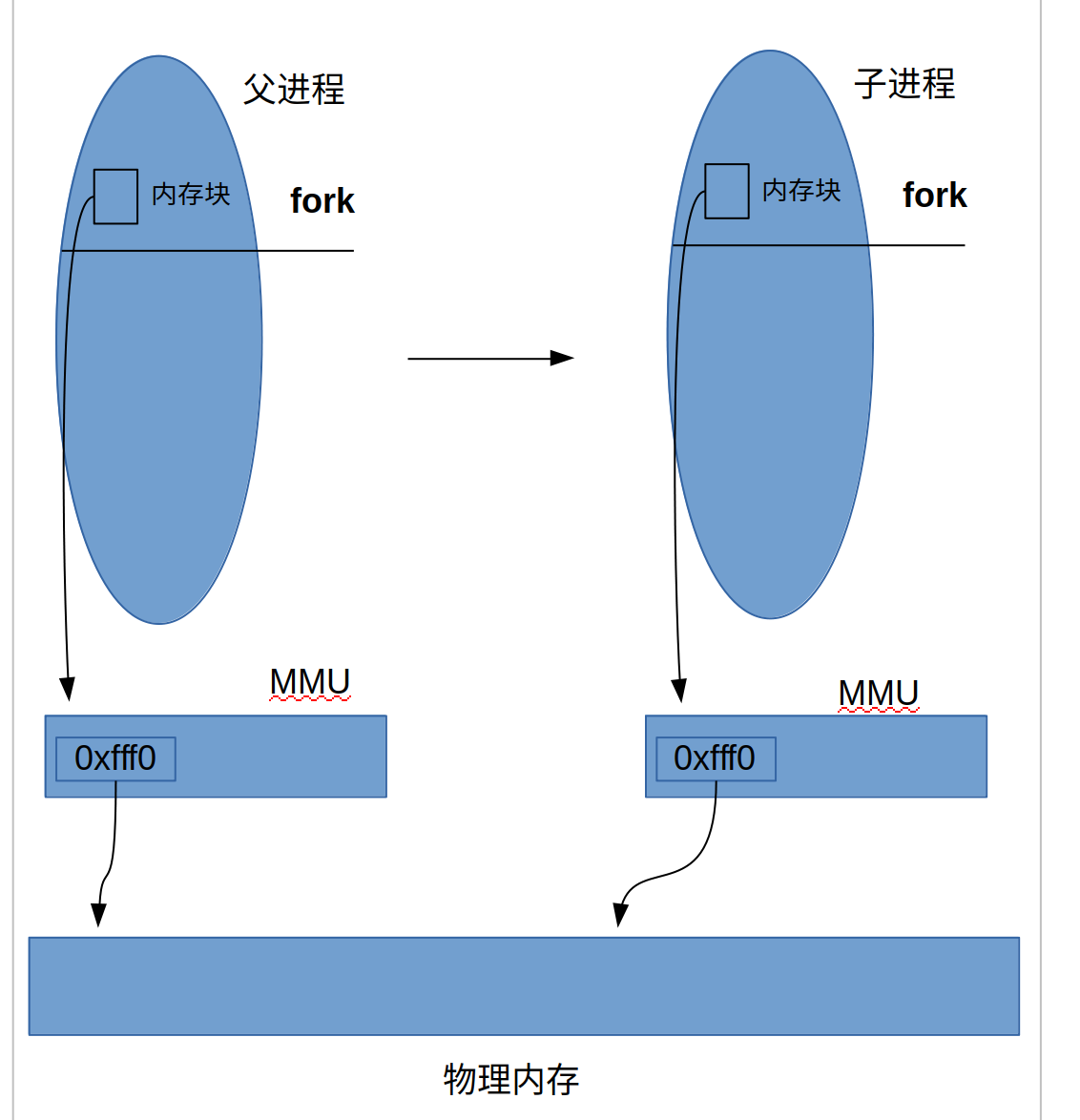
fork后的父子进程 谁先运行与调度策略有关。

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[]);

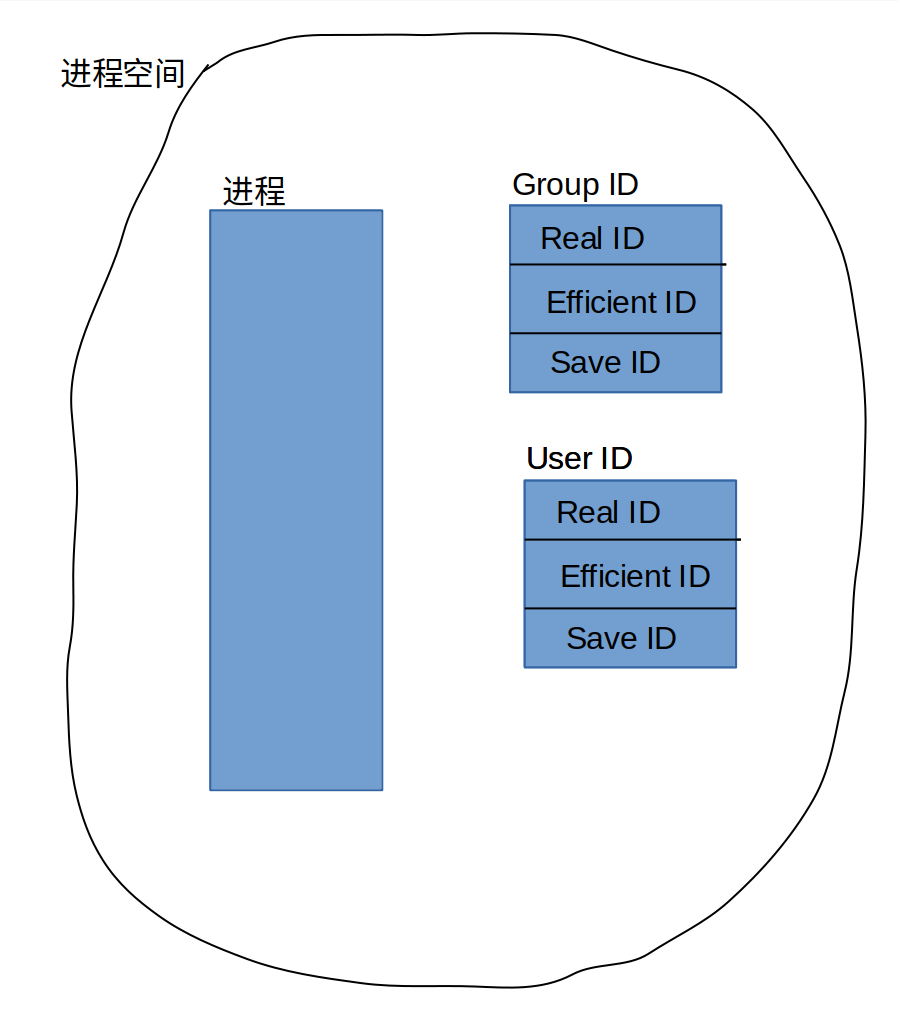
replaces the current process image with a new process image PID不变

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

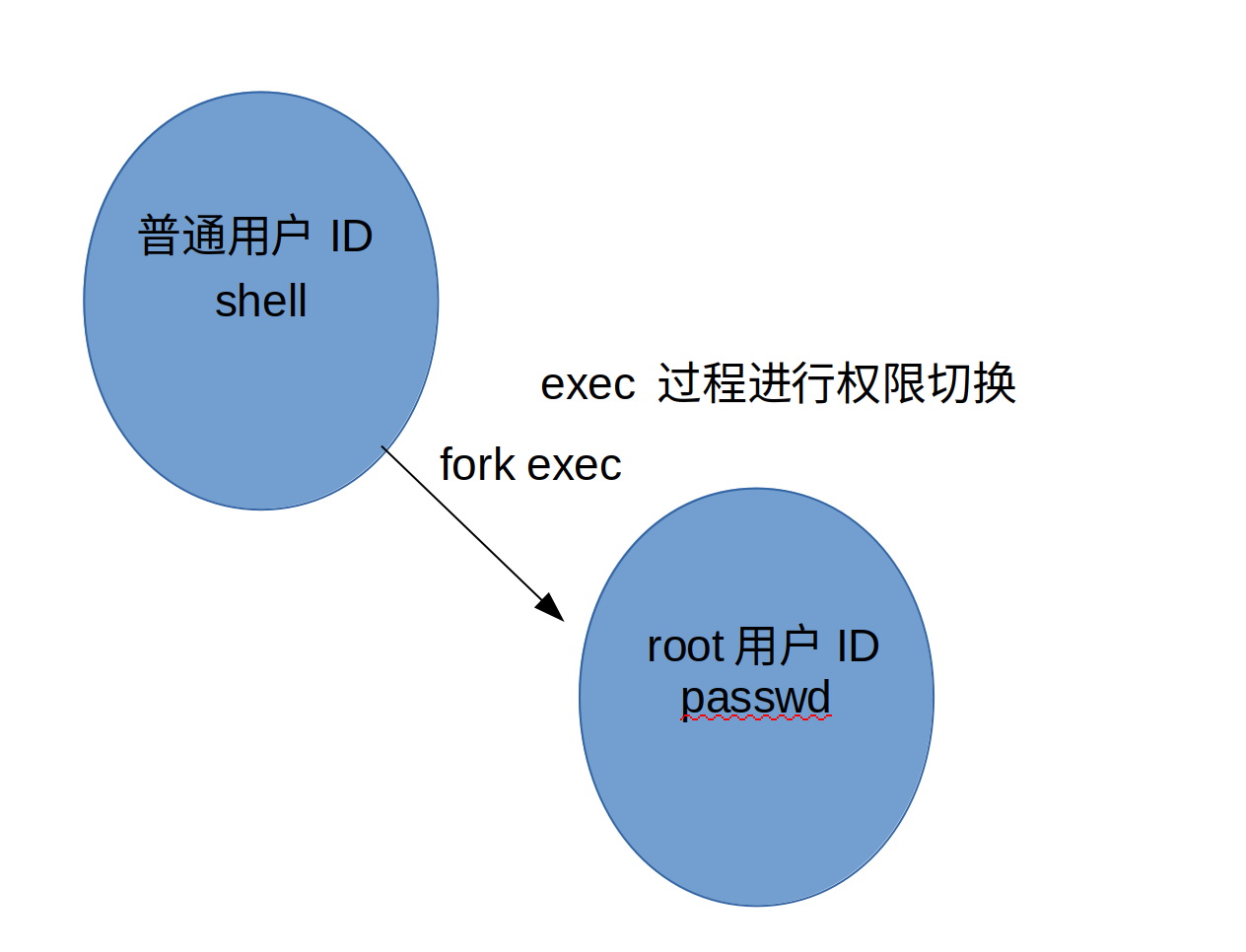
user ID和group ID 都由3个ID组成，rid,eid,sid(有的系统没有)，存在进程空间。



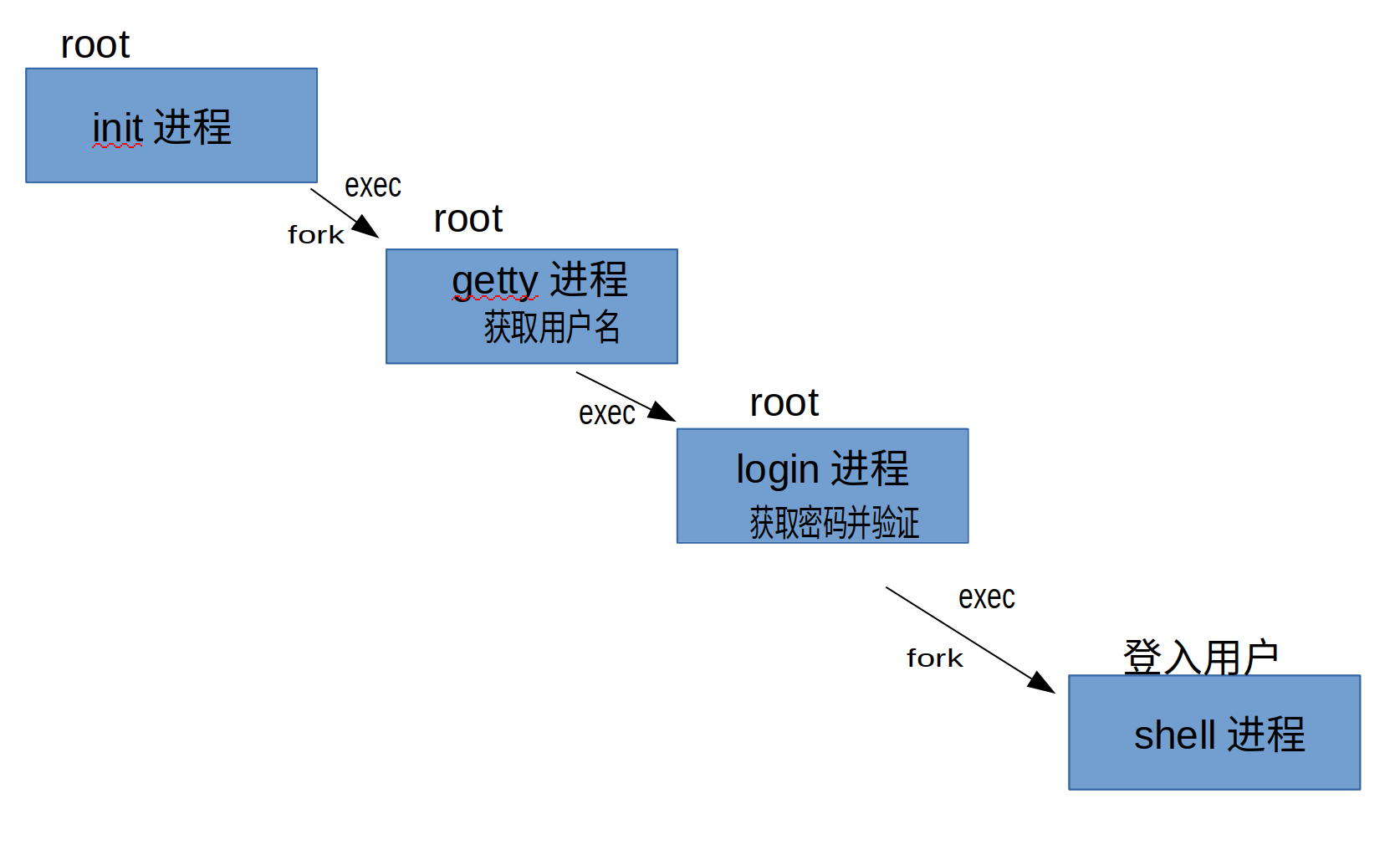
U+S

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



## 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);

## 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

## 进程时间

## 守护进程

## 系统日志