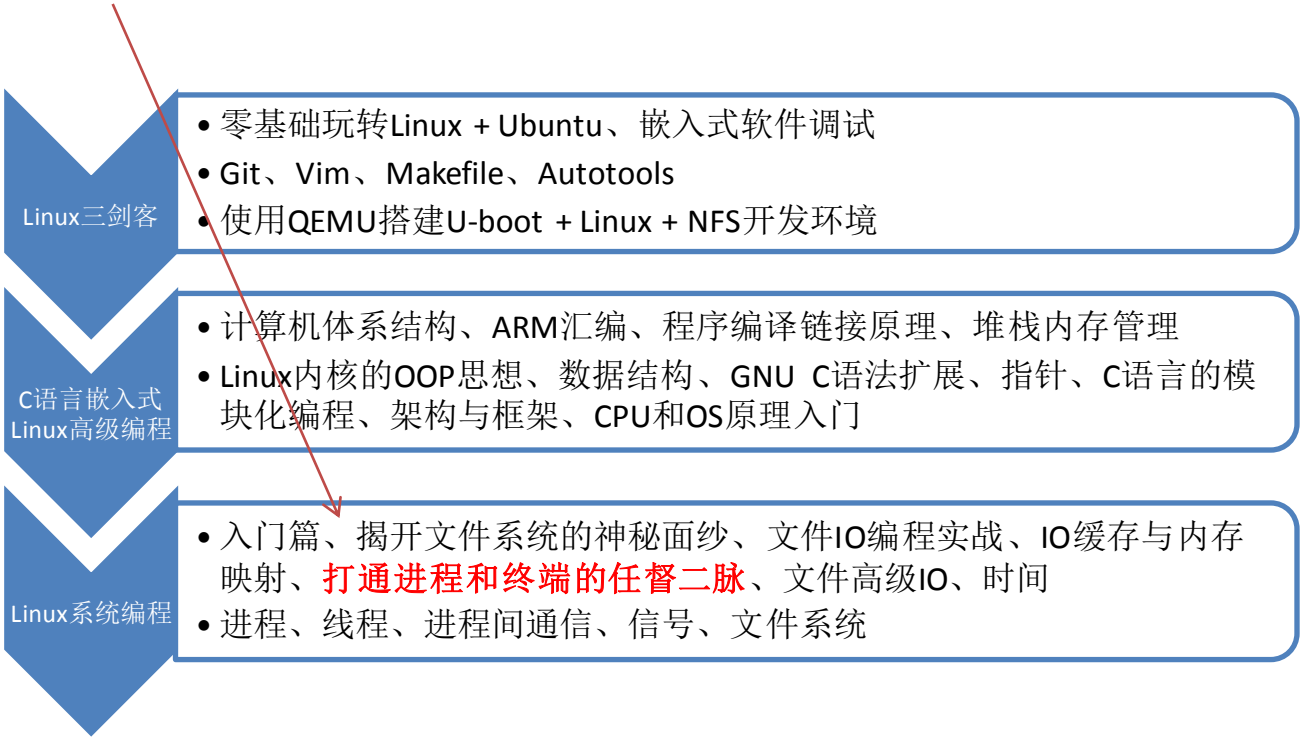
文档版本	说明	作者	创建日期
V0.1	Linux系统编程：入门篇视频配套PPT	王利涛	2018年10月14日
V0.2	第01期：揭开文件系统的神秘面纱	王利涛	2018年11月07日
V0.3	第02期：文件IO编程实战	王利涛	2018年11月25日
V0.4	第03期：IO缓存与内存映射	王利涛	2018年12月11日
V0.5	第04期：打通进程和终端的任督二脉	王利涛	2019年03月15日

《嵌入式工程师自修养》视频教程

- 第00步：Linux三剑客
- 第01步：C语言嵌入式Linux高级编程
- 第03步：Linux系统编程
- 第04步：Linux内核编程
- 第05步：嵌入式驱动开发
- 第06步：项目实战
- ----------
- 详情咨询QQ：3284757626
- 视频淘宝店：wanglitao.taobao.com
- 博客：www.zhaixue.cc
- 微信公众号：

学习路线图

- We are here…



Linux系统编程第04期：

打通进程与终端的任督二脉

程序运行的“牌照”

- 进程与程序的区别
  - 程序：二进制文件，存储在磁盘上
  - 进程：process，一个程序运行实例
    - 将程序从磁盘加载到内存并分配对应的资源、调度运行
  - 进程实例
    - 汇编指令代码、数据、资源、状态
    - 一个虚拟计算机(进程上下文环境、CPU状态寄存器)
    - 进程资源：虚拟内存、打开的文件描述符表、信号、工作目录...

程序运行的“牌照”

- 出租车与“打滴滴”的区别
  - 出租车：一个交通工具，停在马路旁
  - 打滴滴：一个打车运行实例
    - 通过滴滴软件，调度运行，行驶在马路上
    - 需要资源：汽车、司机、汽油、马路、滴滴软件

本期课程学习重点

- 进程的基本概念
- 进程的创建、运行、退出
- 进程的调度、状态
- 进程、进程组、会话
- 进程与终端之间的关系
- 前台进程、后台进程
- 守护进程
- 僵尸进程
- 孤儿进程

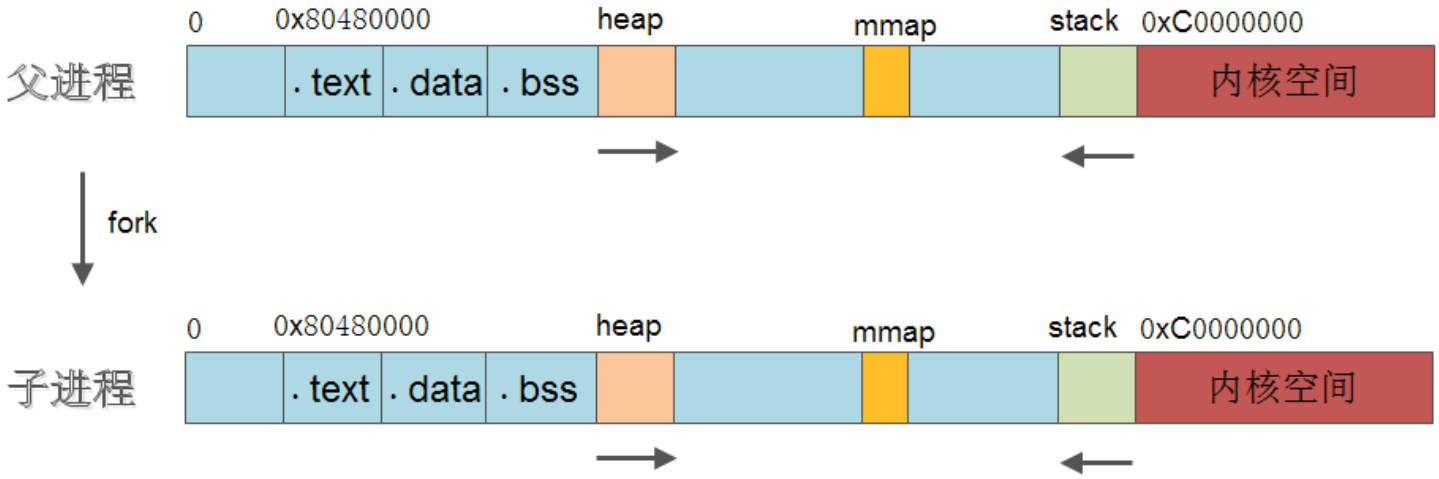
创建一个进程:fork

创建一个子进程

- 系统调用：fork()
  - 函数原型：pid_t fork(void);
  - 函数作用：创建一个新进程
  - 返回值：
    - -1 ：创建子进程失败
    - 0 ：在子进程中返回0
    - >0：在父进程中返回的是子进程的PID

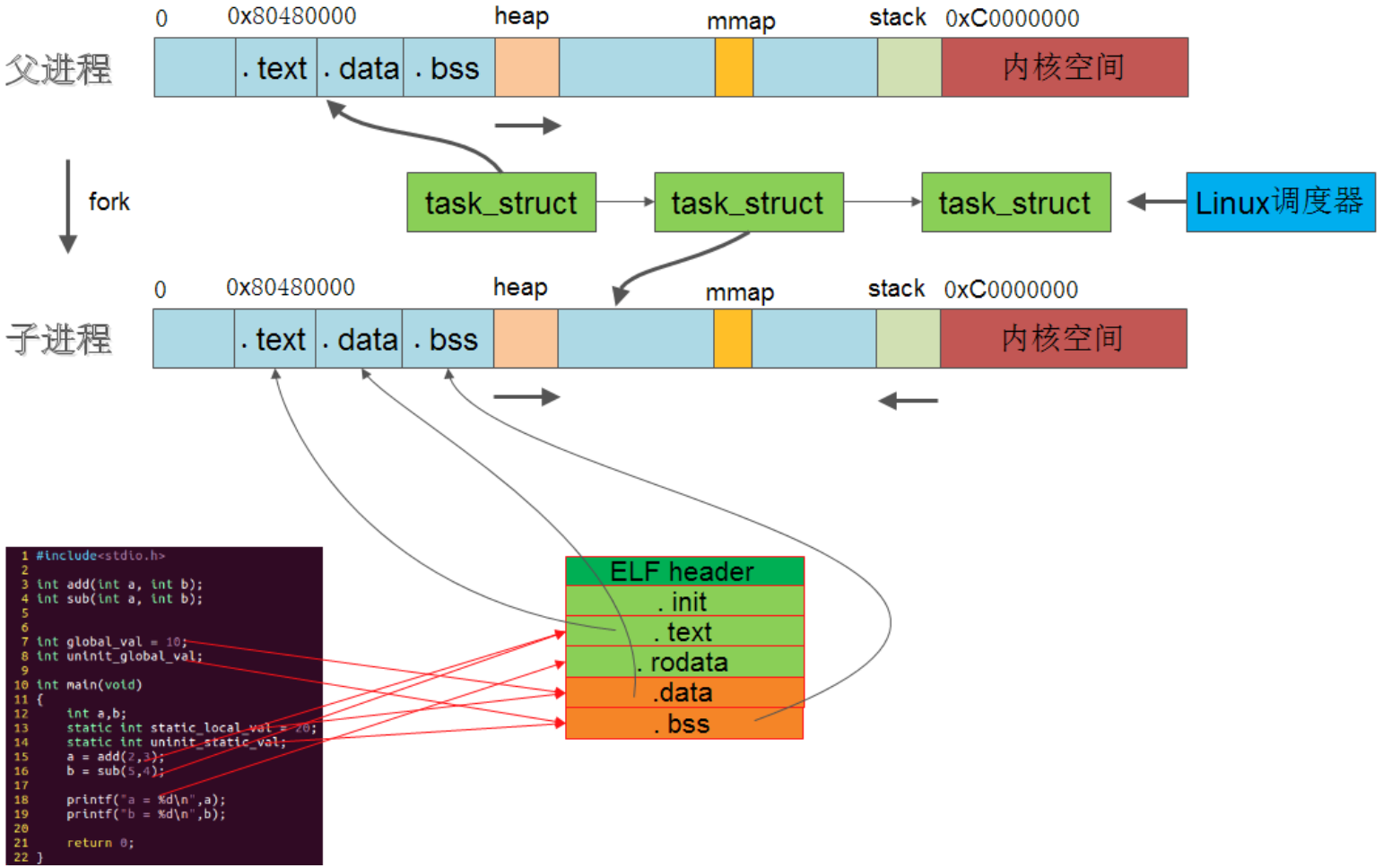
子进程的运行

- 子进程拷贝父进程
  - 代码、数据、堆栈内存
  - 进程资源：打开的文件描述符、信号、缓冲区...



子进程的运行：“借壳上市”

执行一个二进制程序文件



执行一个二进制程序文件

- execvp函数
  - 函数原型：int execvp (const char *file, char *const argv[]);
  - 功能说明：将当前进程的代码使用file程序文件代替并执行
  - 参数说明
    - file：要执行的程序名称
    - argv：要执行的程序文件的参数列表，参数列表以NULL指针为结束标记
  - 返回值
    - 成功：无返回值
    - 失败：返回-1，并设置errno值

编程作业：实现一个mini shell

- Shell执行二进制文件过程：
  - Shell交互环境：用户输入命令、参数
  - Shell解析命令、参数，调用fork创建一个子进程
  - 调用exec函数，载入命名程序到内存，替换掉子进程的代码
  - 将解析的参数列表argv传给main()入口函数
  - 执行main()函数

执行一个二进制程序文件

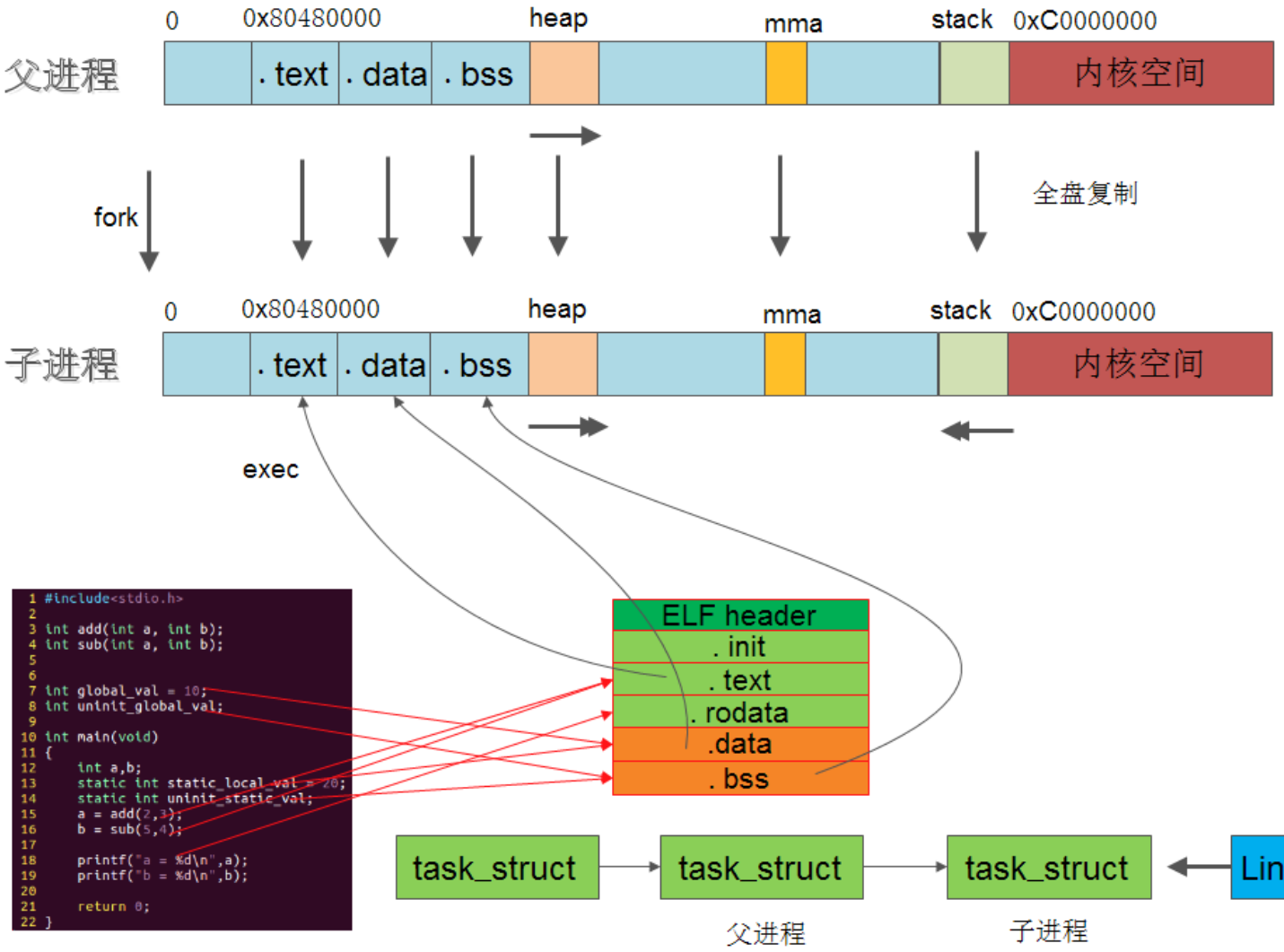
- exec函数簇
  - #include <unistd.h>
  - int execl (const char *path, const char *arg, ...);
  - int execlp (const char *file, const char *arg, ...);
  - int execle (const char *path, const char *arg, ...);
  - int execv (const char *path, char *const argv[]);
  - int execvp (const char *file, char *const argv[]);
  - int execvpe (const char *file, char *const argv[], char *const envp[]);

执行一个二进制程序文件

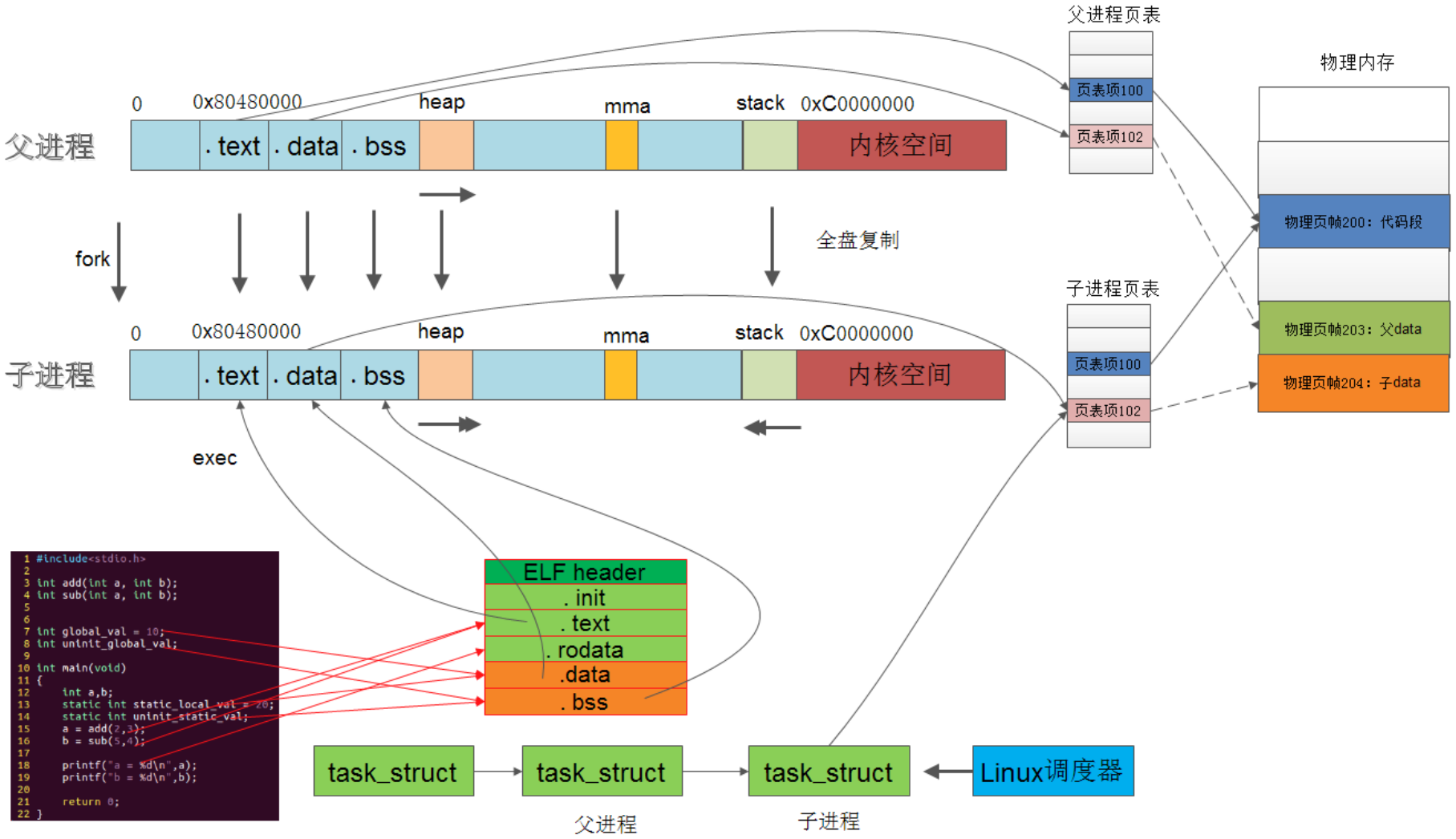
- exec函数簇命名规则
  - L：参数以列表的形式提供
  - V：参数以数组(向量)的方式提供
  - E：为新进程提供新的环境变量
  - P：在用户的绝对路径path下查找可执行文件，该文件必须在用户路径下，可以只指定程序文件名

写时复制(COW)与vfork

一个新进程的诞生：虚拟空间

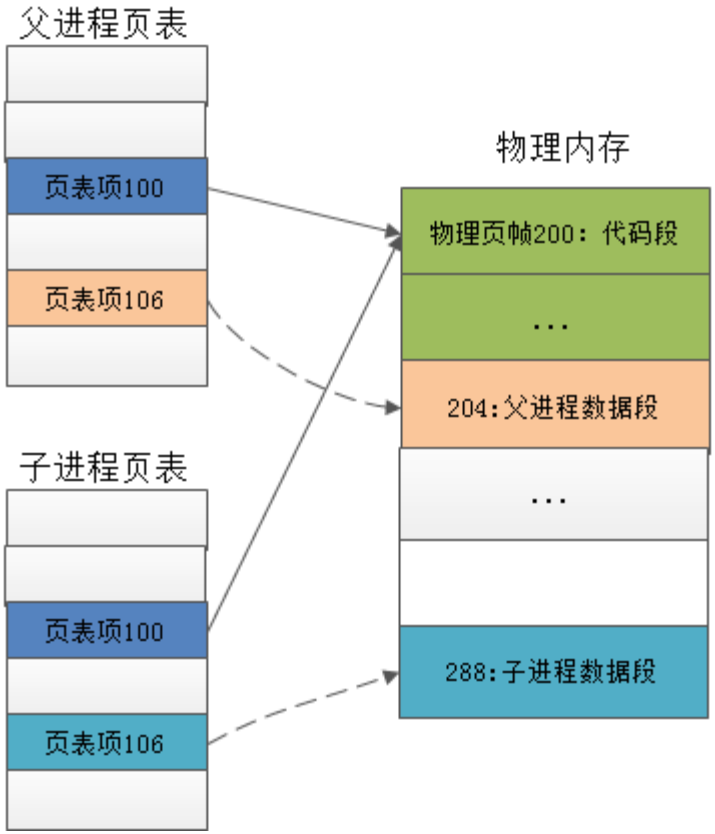


一个新进程的诞生：物理空间




```c
#include<stdio.h>

int add(int a, int b);
int sub(int a, int b);


int global_val = 10;
int uninit_global_val;

int main(void)
{
    int a,b;
    static int static_local_val = 20;
    static int uninit_static_val;
    a = add(2,3);
    b = sub(5,4);

    printf("a = %d\n",a);
    printf("b = %d\n",b);

    return 0;
}
```

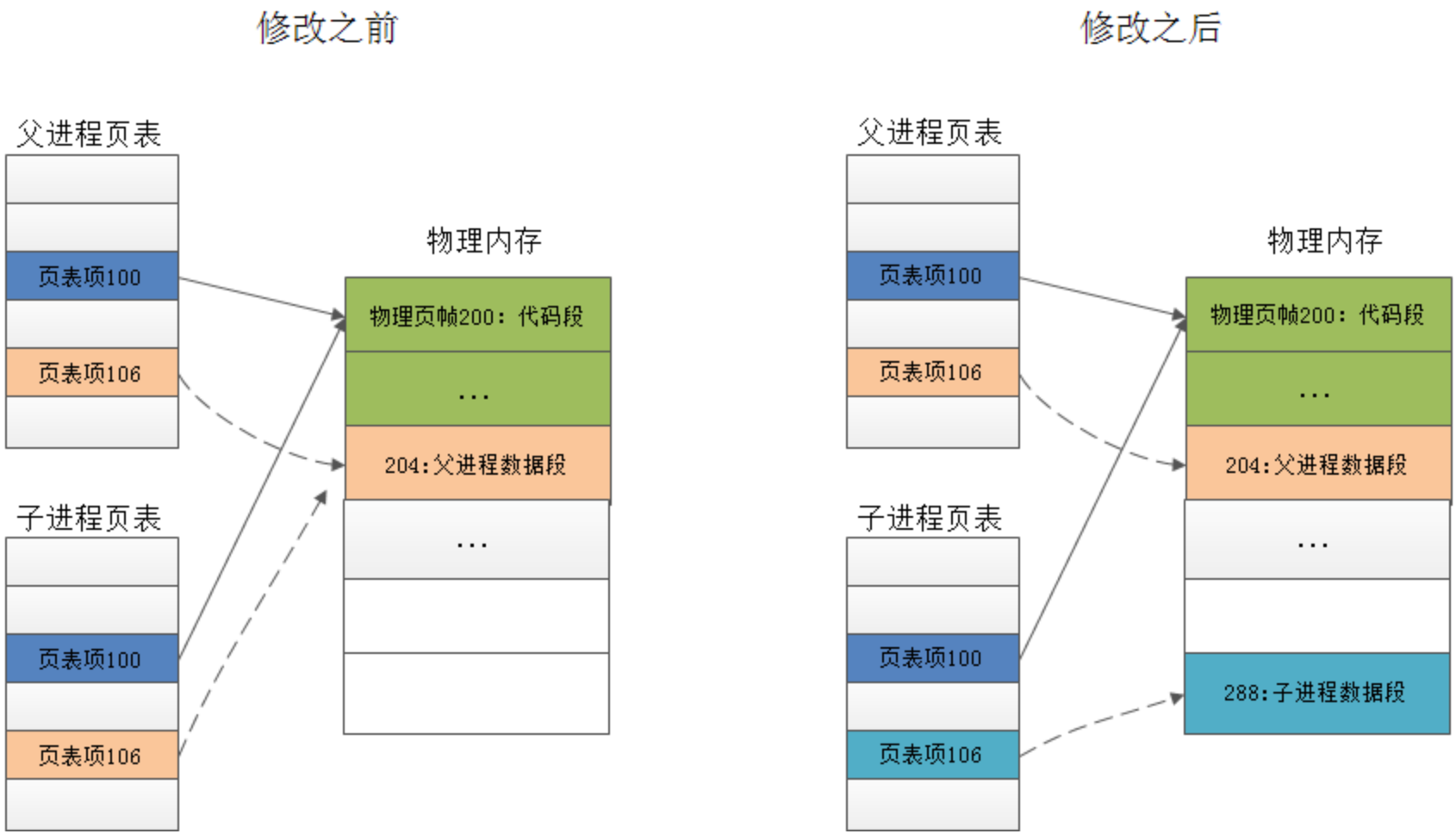
一个新进程的诞生


父进程页表
页表项100
页表项106
子进程页表
页表项100
页表项106
物理内存
物理页帧200：代码段
...
204:父进程数据段
...
288:子进程数据段

写时复制(copy-on-write)

- 对fork-exec流程的改进
  - 对于代码段、数据段等，父子进程可以共享，节省拷贝开销
  - 父子进程的页表项均指向同一块物理内存页帧
  - 当子进程进程空间的内容要修改时，才会真正将段复制到子进程
  - 写时复制：
    - 仅仅为子进程复制父进程的虚拟页表项
    - 对将要修改的页面修改页表项

写时复制(copy-on-write)



系统调用：vfork

- 对fork的改进
  - 对fork的改进更为彻底、简单粗暴
  - vfork是为子进程立即执行exec的程序而专门设计的
    - 无需为子进程复制虚拟内存页或页表，子进程直接共享父进程的资源，直到其成功执行exec或是调用exit退出
    - 在子进程调用exec之前，将暂停执行父进程

进程的退出：exit

终止当前进程

- exit函数
  - POSIX标准和ANSI C定义的标准函数
    - #include <stdlib.h>
    - 其实是对系统调用_exit的封装
  - 函数原型：void exit (int status);
  - 函数功能：终止当前进程
  - 参数说明：用于标识进程的退出状态，shell或父进程可以获取该值
    - 0：表示进程正常退出
    - -1/1：表示进程退出异常
    - 2~n：用户可自定义

exit函数背后

- 执行流程
  - 调用退出处理程序(通过atexit、on_exit注册的函数)
  - 刷新stdio流缓冲区
  - 使用由status提供的值执行_exit系统调用函数
    - 关闭进程打开的文件描述符、释放进程持有的文件锁
    - 关闭进程打开的信号量、消息队列
    - 取消该进程通过mmap创建的内存映射
    - …

atexit/on_exit

- 退出处理程序
  - 在exit退出后可以自动执行用户注册的退出处理程序
  - 执行顺序与注册顺序相反
  - 函数原型：int atexit (void (*function)(void));
  - 函数原型：int on_exit (void (*function)(int , void *), void *arg);

TIPS

- return与exit的区别
  - exit用来终止当前进程，将控制权交给操作系统
  - return用来退出当前函数，销毁栈帧，返回到上级函数执行
  - 终止进程：
    - 正常退出：exit、_exit、从main函数return
    - 异常退出：调用abort、信号ctrl + C

思考

- main()函数可以通过return或exit终止进程
- main()函数return后，为什么相当于调用exit()，进程就退出了？

编译器会加上exit

TIPS

- exit_group函数
  - 函数原型： void exit_group (int status);
  - exit：退出当前进程process
  - exit_group：退出一个进程中所有threads
  - Linux系统特有的系统调用，不属于POSIX标准

TIPS

- fork之后、exec之前，使用exit是不安全的
- 很多资源还是共享的(如文件描述符、缓冲区)

进程的退出：exit与_exit

_exit和exit

- 两者的区别
  - exit是库函数是对_exit系统调用的封装
  - 在调用_exit之前，它会执行各种动作
    - 调用退出处理程序(通过atexit和on_exit注册的回调函数)
    - 刷新stdio流缓冲区
    - 使用由status提供的值执行_exit系统调用

_exit和_Exit

- _exit的执行流程
  - 关闭进程打开的文件描述符、释放该进程持有的文件锁
  - 关闭该进程打开的信号量、消息队列
  - 取消该进程通过mmap()创建的内存映射
  - 将该进程的所有子进程交给init托管
  - 给父进程发送一个SIGCHLD信号
  - ……

小结

- 关于_exit和exit总结
  - 在一个进程中，直接调用_exit终止进程，缓冲区的数据可能会丢失
  - 在创建子进程的应用中，只应有一个进程(一般为父进程)调用exit终止，而其他进程应调用_exit()终止。从而确保只有一个进程调用退出处理程序并刷新stdio缓冲区
  - 如果一个进程使用atexit/on_exit注册了退出管理程序，则应使用exit终止程序的运行，否则注册的回调函数无法执行

TIPS

- 终止进程的各种方法
  - 从main函数return
  - 调用库函数：exit
  - 调用系统调用：_exit/_Exit
  - 调用abort：_exit的内部实现(POSIX)
  - 信号：Ctrl+C
  - …

进程的退出：vfork与exit

fork函数的改进

- fork函数的开销
  - 子进程对父进程数据、堆栈、缓存等资源的拷贝
  - 子进程exec函数的执行
  - 写时复制：copy-on-write：先不复制，需要修改的时候再拷贝

vfork函数

- 系统调用vfork
  - 子进程共享父进程的代码、数据、堆栈资源
  - 使用vfork后，直接运行exec，节省了资源拷贝的时间
  - 使用vfork，创建子进程后直接运行子进程、父进程被阻塞

TIPS

- 使用vfork创建子进程的退出
  - vfork 创建的子进程共享父进程的代码段、数据段、堆栈，子进程退出时使用_exit/exit，使用return会破坏父进程的堆栈环境、产生段错误
  - 父进程退出一般使用exit，而子进程退出使用_exit
    - 子进程exit，输出不确定，依赖于IO库的实现[APSE]
    - 子进程：_exit + fflush = exit ?

等待子进程终止：wait

等待子进程运行终止

- wait()函数
  - Void exit(int status);
  - 函数原型： pid_t wait(int *status);
  - 函数功能：等待子进程的终止及信息
  - 参数说明：子进程调用exit/_exit时的status
  - 返回值
    - Wait调用成功，会返回已终止子进程的pid
    - Wait调用失败，返回-1，设置errno值
    - 若子进程没有终止，wait调用会阻塞父进程，直到子进程终止，子进程终止后，该调用立即返回

子进程的返回状态

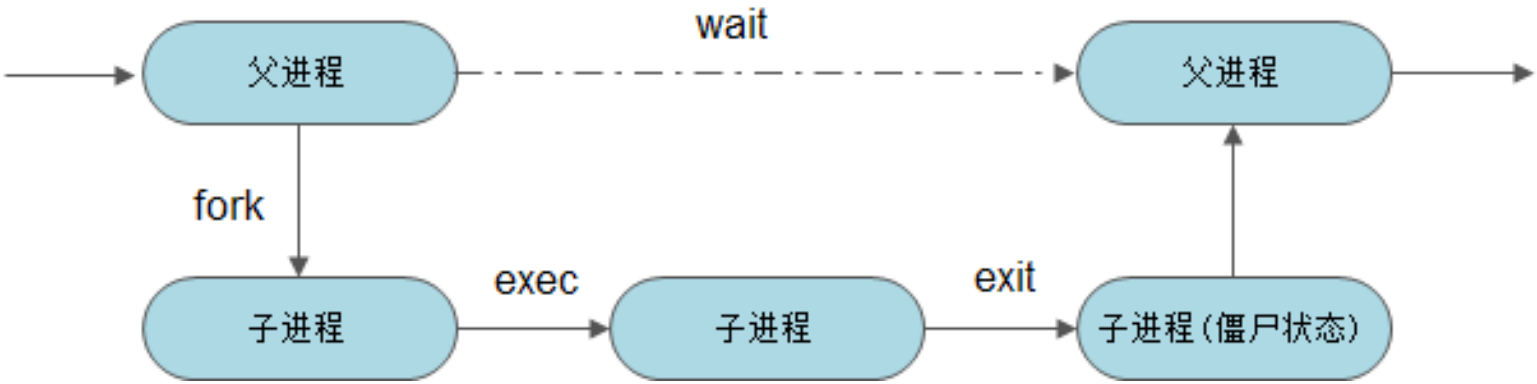
- 子程序的返回状态
  - 定义在：wait.h

图 26-1：自 wait()和 waitpid()的 status 参数所返回的值

子进程的返回状态

- 通过宏来解析返回状态
  - WEXITSTATUS(status)：返回子进程的退出状态
  - WTERMSIG(status)：子进程因未捕捉的信号而终止，此宏返回true
  - WSTOPSIG(status)：子进程因信号暂停，此宏返回true
  - WIFEXITED(status)：若子进程正常结束，返回true
  - WIFSIGNALED(status)：若通过信号杀掉子进程，此宏返回true
  - WIFSTOPPED(status)：若子进程因信号而停止，此宏返回true
  - WIFCONTINUED(status)：若子进程收到SIGCONT恢复运行，返回true

小结

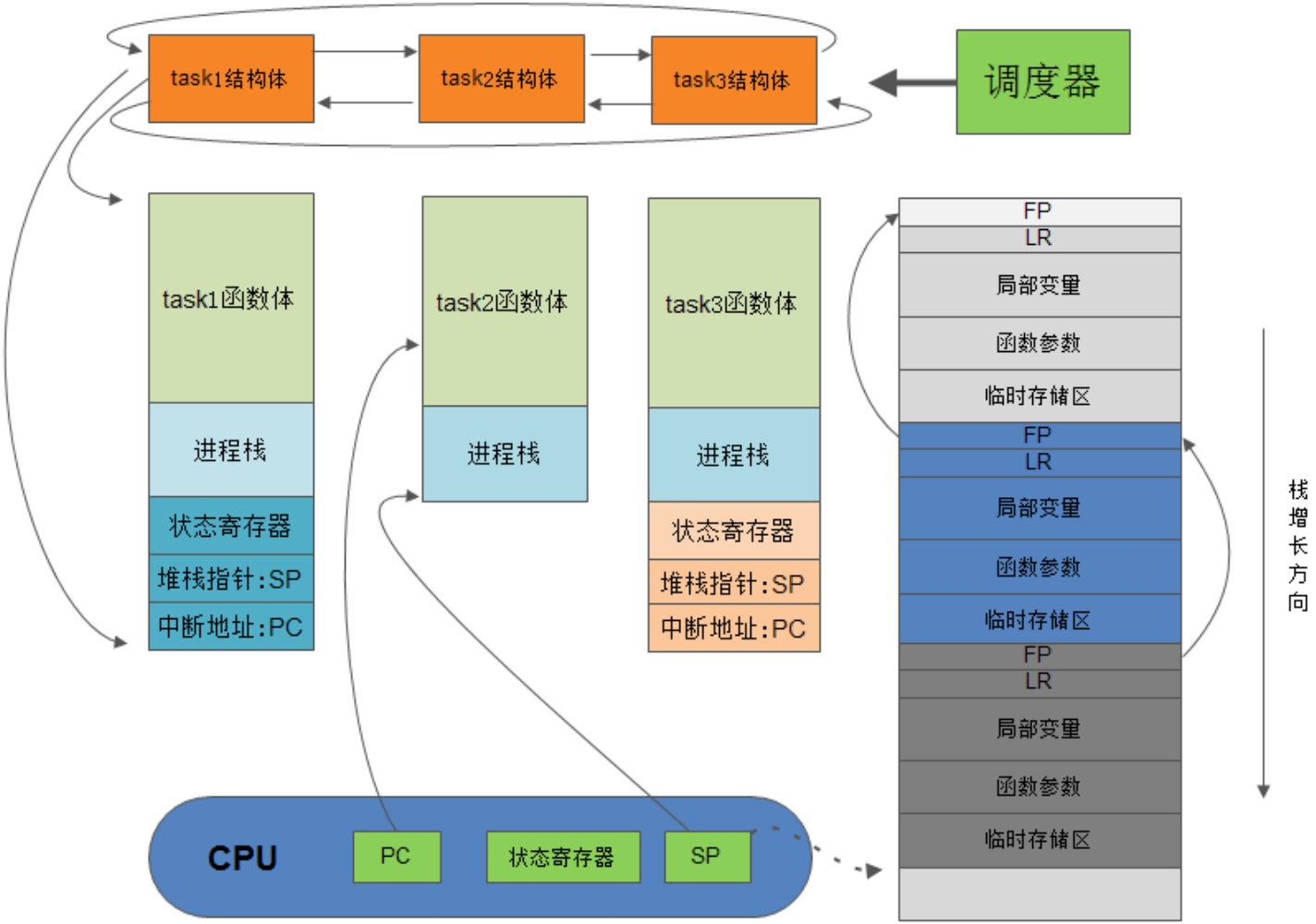


等待特定子进程运行终止

- waitpid()函数
  - 函数原型： pid_t wait(pid_t pid, int *status, int options);
  - 函数功能：等待特定子进程的终止及信息
  - 参数说明：子进程调用exit/_exit时的status
  - 返回值
    - Wait调用成功，会返回已终止子进程的pid
    - Wait调用失败，返回-1，设置errno值
    - 若子进程没有终止，wait调用会阻塞当前进程，直到子进程终止，子进程终止后，该调用立即返回
- waitid()函数
  - System V 系统调用接口
  - 函数原型： int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
  - idtype：P_ALL等待任何子进程；P_PID、P_PGID等待特定进程/进程组

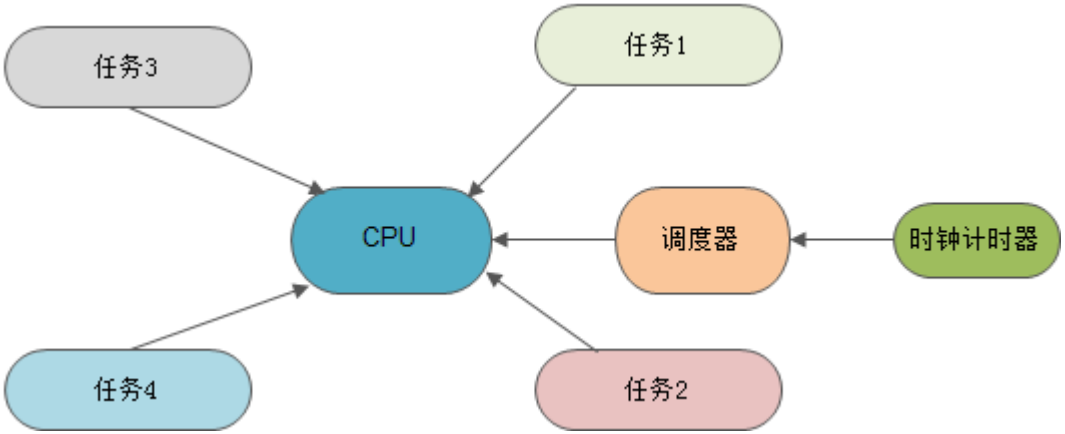
进程调度

操作系统的核心：任务管理



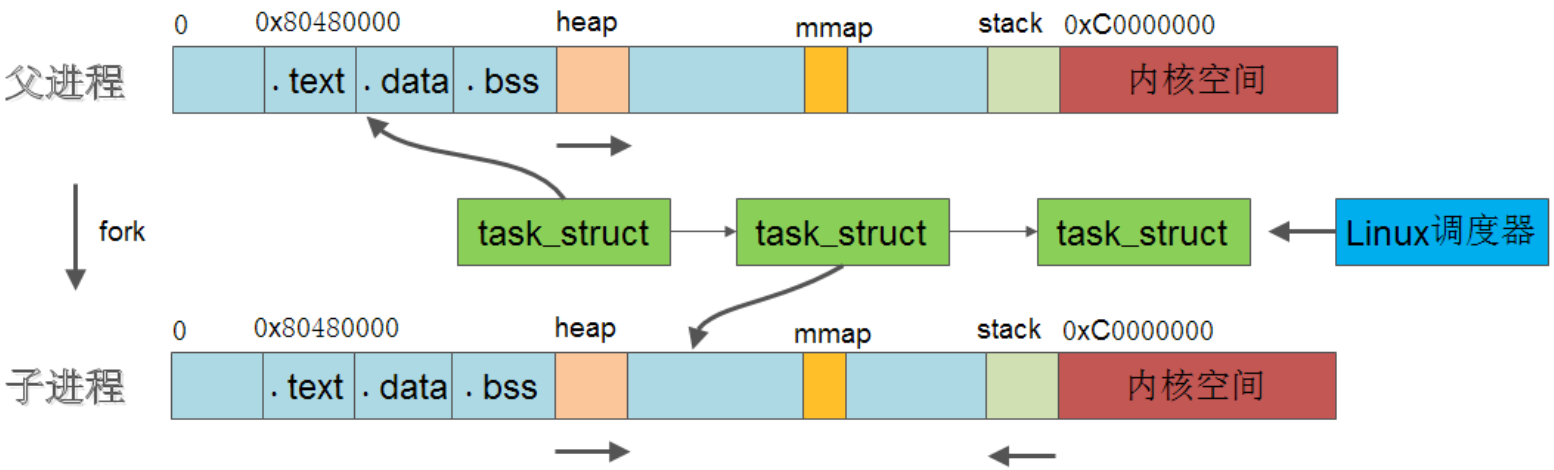
调度器

- 任务调度
  - 将有限的CPU资源分配给多个进程
  - 目的：最大化处理器效率，让多个进程同时运行、互不影响
  - 实现：
    - 协同式：一个进程运行完自己的时间片，主动退出，CPU无权过问
    - 抢占式：时间片到了或有更高优先级、调度器抢占CPU进行任务切换



Linux进程管理

- 进程分类
  - 处理器消耗型
    - 渴望获取更多的CPU时间，并消耗掉调度器分配的全部时间片
    - 常见例子：无限死循环、科学计算、影视特效渲染
  - I/O消耗型
    - 由于等待某种资源通常处于阻塞状态，不需要较长的时间片
    - 常见例子：等待用户输入、GUI程序、文件读写I/O程序



Linux进程管理

- Linux调度策略
  - 对不同进程采取不同调度策略、实现多个调度器
    - 完全公平调度CFS
    - 实时进程调度RT
    - 最终期限调度DL
    - IDLE类调度器、STOP类调度器
  - 不同进程由不同的调度器管理，彼此之间互不干扰
    - 处理器消耗型进程：减少优先级、分配尽可能长的时间片
    - I/O消耗进程：增加优先级、增加实时性、增强用户体验
    - 两者混合型

进程的优先级

- 进程的nice值和优先级
  - $ nice -n 5 top
  - NI [-20,19] ：进程的NICE值，也叫静态优先级，nice值越小，抢占CPU能力越强，nice会影响进程的优先级
  - PRI[0,139]：进程的优先级，也叫动态优先级，值越小，优先级越高
  - 进程默认优先级： 20

    #define DEFAULT_PRIO (MAX_RT_PRIO + NICE_WIDTH / 2)
  - 实时进程与非实时进程
    - 实时进程：优先级[0,99]，采用实时进程的调度算法
    - 非实时进程：优先级[100,139]，采用O1/CFS等调度算法

TIPS

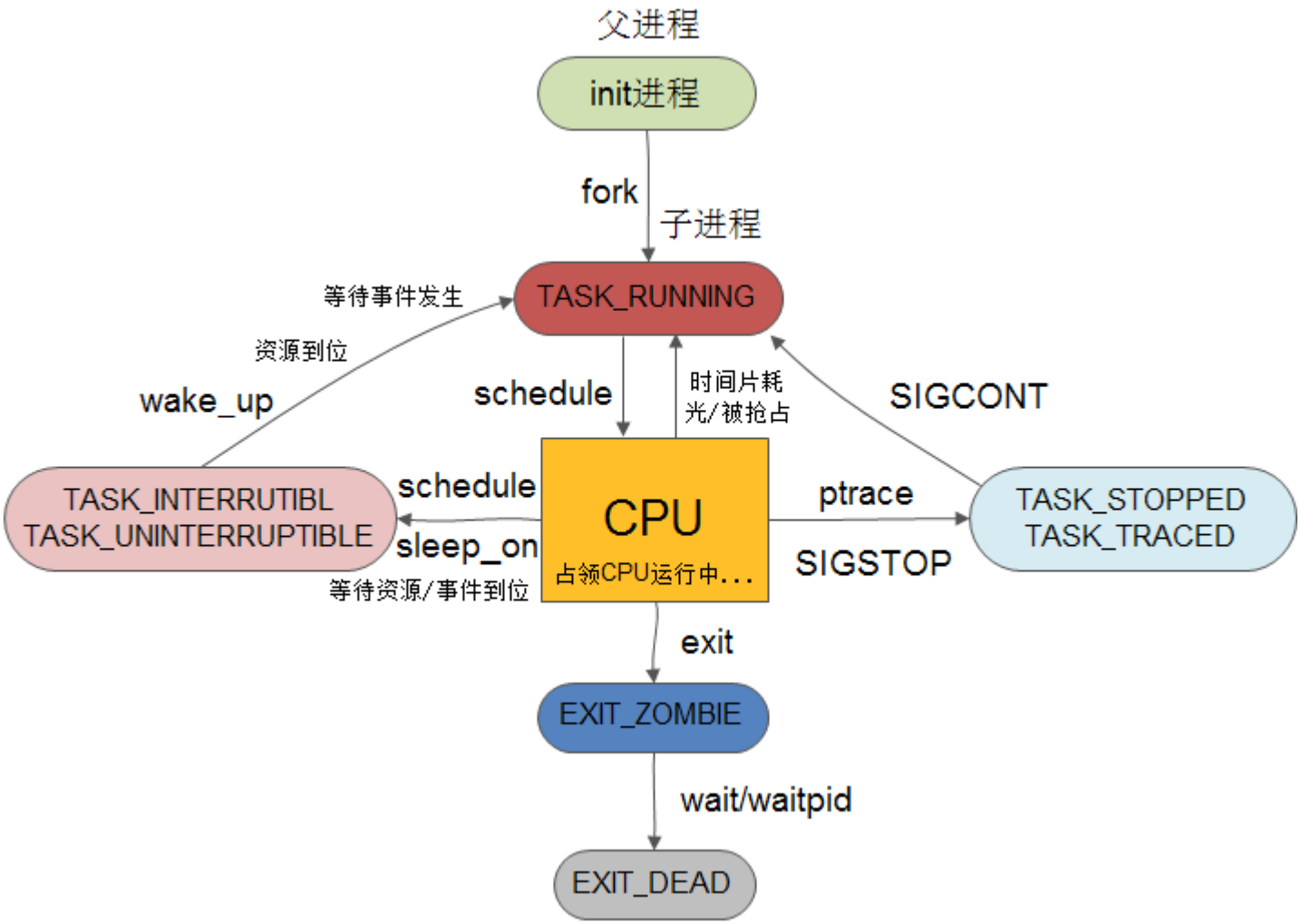
- 并发与并行区别
  - 并发：concurrency，CPU通过时间片轮转同时做多件事情
  - 并行：parallellism，很多事情在多个CPU上同时进行
  - 并发可以看做并行的一个“子集”
  - 一个应用程序
    - 可以是并发的，但不是并行的
    - 可以是并行的，但不是并发的
    - 既是并发的，又是并行的
    - 既不是并发的，又不是并行的

Linux进程状态

Linux进程状态

- 进程状态
  - TASK_RUNNING：就绪/可运行状态
  - TASK_INTERRUPTABLE：进程被挂起(睡眠)，直到等待条件为真被唤醒
  - TASK_UNINTERRUPTABLE：深度睡眠，睡眠期间不响应信号
  - TASK_STOPPED：进程的执行被暂停
  - TASK_TRACED：被其它进程跟踪，常用于调试
  - EXIT_ZOMBIE：僵死状态，进程的执行被终止
  - EXIT_DEAD：僵死撤销状态，防止wait类系统调用的竞争状态发生

Linux进程状态


父进程
init进程
fork
子进程
TASK_RUNNING
等待事件发生
资源到位
wake_up
schedule
时间片耗光/被抢占
SIGCONT
TASK_INTERRUTIBL
TASK_UNINTERRUPTIBLE
schedule
sleep_on
等待资源/事件到位
CPU
占领CPU运行中...
ptrace
SIGSTOP
TASK_STOPPED
TASK_TRACED
exit
EXIT_ZOMBIE
wait/waitpid
EXIT_DEAD

Linux进程状态

- 查看进程状态:$ ps ax
  - S：进程呈睡眠态，通常等待某个事件，如一个信号
  - R：就绪/可运行状态
  - D：深度睡眠，即不可中断的睡眠，通常指等待输入或输出完成
  - T：进程被暂停执行，如进程被shell的ctrl+z 暂停，或处于调试中
  - Z：僵尸进程
  - N：低优先级进程
  - s：进程是会话首进程
  - +：进程属于前台进程组
  - l：进程是多线程的(#注:小写的L)
  - L：has pages locked into memory (for real-time and custom IO)
  - <：高优先级进程

进程的UID和GID

进程的PID

- 进程的PID
  - 创建进程时系统为每个进程自动分配一个整数，用于进程身份识别
  - 每个进程的PID在内核中是唯一的，不能重复
  - PID资源有限：$ cat /proc/sys/kernel/pid_max
  - PPID：该进程父进程的ID号
  - $ ps ：显示当前终端由当前用户运行的进程
  - $ ps –a：显示所有终端所有用户正在运行的进程

用户和组

- 用户和用户组的关系
  - Linux是多用户操作系统，可以允许不同的用户登录
    - 每个用户可以属于不同的组，实现资源的访问权限控制
    - 创建用户没有指定所属组时，系统会创建一个跟用户同名的组
    - 每个用户、组用一个整数ID标识 (uid_t和gid_t类型，其实就是u32)
    - UID：user ID，创建用户时系统分配的一个整数号码
    - GID：group ID，系统给用户组分配的一个整数号码
  - /etc/passwd 和/etc/group文件
    - Linux系统通过 /etc/passwd 和/etc/group文件将ID与用户名建立关联
    - 如 root用户的UID为0，GID为0 | wit 用户的UID为1000，GID为1000
    - 该文件还包括的信息
      - » 组ID：用户所属第一个组的整数型组ID
      - » 主目录：用户登陆后所居于的初始目录
      - » 登录shell：执行以解释用户命令的程序名称

用户和组

- Linux系统中的UID分配
  - 0~255：特权用户
  - 1000以上：普通用户

进程UID和GID

- 进程与UID之间的关系
  - 每一个进程与用户、用户组相关联
    - 进程必须以合适的用户和组运行
    - 运行进程的用户ID即该进程的UID
    - 该UID和GID表示进程的资源访问、操作权限
  - 进程的各种ID
    - RUID：实际用户ID。运行该进程的那个登录用户ID，一般会继承父进程，子进程可通过setuid修改
    - EUID：有效用户ID。当前进程所使用的用户ID，一般等于RUID，用于权限验证。
    - SUID：设置用户ID。父进程原先的有效ID(执行setuid之前)，子进程会继承并将其设置为有效用户ID

编程作业

- 创建一个进程，修改进程的UID并打印出来
- 相关API
  - setuid：改变当前进程的UID
  - getuid：获取当前进程的UID
  - seteuid：改变当前进程的有效UID
  - geteuid：获取当前进程的有效UID

通过proc查看进程资源

进程资源

- 结构体：task_struct

```
struct task_struct {
        void            *stack;
        atomic_t        usage;
        const           struct sched_  class *sched_class;
        struct          list_head       tasks;
        struct          mm_struct  *mm, *active_mm;
        struct          vm_area_struct *vmacache[VMACACHE_SIZE];
        int             exit_state;
        pid_t           pid;
        pid_t           tgid;
        struct          task_struct  *parent;
        struct          list_head      children;
        struct          fs_struct      *fs;
        struct          files_struct   *files;
        struct          bio_list       *bio_list;
        struct          reclaim_state  *reclaim_state;
        ...
        int             pagefault_disabled;
        struct          thread_struct               thread;
}
```

通过ps命令查看进程资源

- ps命令
  - -A/-e：显示系统所有的进程(包括守护进程)，相当于-e
  - -a：显示所有终端下的所有用户运行的进程
  - -u：显示用户名、CPU百分比和内存的使用
  - -x/-f：列出进程的详细信息
  - -H：显示进程树
  - -r：只显示正在运行的进程
  - -o：分类输出
  - $ ps –o pid,ppid,state,tty,command
  - 监视进程：$ top

通过proc查看进程资源

- procfs文件系统
  - Linux内核：一切皆文件，ps命令也是从proc文件读取数据
  - Procfs是Linux内核中一个特殊的文件系统，以/proc目录形式呈现
  - 应用程序可通过/proc下的文件接口对驱动和内核信息进行访问
    - 获取内核信息、硬件设备信息、进程信息
    - 设置内核参数、控制开关

进程资源

- 通过proc查看进程资源
  - 进程对应的可执行文件名字
  - 环境变量、CPU、内存相关信息
  - 进程的上下文环境、堆栈
  - 进程状态
  - 进程打开的文件列表

进程资源

- 通过proc查看进程资源

文件	说明
cmdline	以\0分割的命令行参数
cmd	指向当前工作目录的符号链接
environ	NAME=value键值对环境列表，以\0分隔
exe	指向正在执行文件的符号链接
fd	文件目录
maps	内存映射
mem	进程虚拟内存
mounts	进程的安装点
root	指向根目录的符号链接
status	进程的各种信息：PID、内存使用量、信号
task	进程中的每个线程均包含一个子目录

与进程通信：信号

与进程通信

- 信号
  - 信号是一种异步通信的IPC
  - 可以给一个指定进程发送一个信号
  - 进程根据接收信号类型作相应的处理
  - 系统调用接口：signal、kill

一个进程对信号的处理

- 三种处理方式
  - 如果注册信号处理回调函数的话，会调用注册的信号处理回调函数
  - 如果没有注册的话，按照该信号在系统中的默认处理方式
    - 忽略
    - 终止进程

信号与其对应的系统事件

信号系统	信号对应动作	说明
SIGHUP(1)	进程终止	控制终端被关闭时产生
SIGINT(2)	进程终止	按键产生中断信号如：ctrl + c
SIGQUIT(3)	进程终止并转储文件	从键盘按键产生的退出信号：ctrl+\
SIGSEGV(11)	进程终止并转储文件	访问非法内存时产生该信号
SIGKILL(9)	进程终止	Kill命令产生的信号
SIGCHLD(17)	自定义处理	子进程暂停或终止时产生
SIGSTOP/CONT	进程暂停/恢复运行	系统暂停/恢复信号

终端与信号

- 终端驱动支持的信号
  - CTRL + Z：SIGTSTP
  - CTRL + \：SIGQUIT
  - CTRL + C：SIGINT

终端与控制台

终端的概念

- 什么是终端？
  - 英文terminal，计算机外围设备，用来处理用户信息输入和结果输出
  - 终端本身无计算能力，只是一个连接设备(如通过串口连接)

终端的演变

- TTY设备的演变
  - 电传打印机(Teletype)：早期的终端，由键盘、打印设备构成，简称TTY设备
  - 大型机/小型机：多用户登录，显示器太贵，将电传打印机作为终端，通过串口连接到主机，通过用户名和密码登录主机

终端的演变

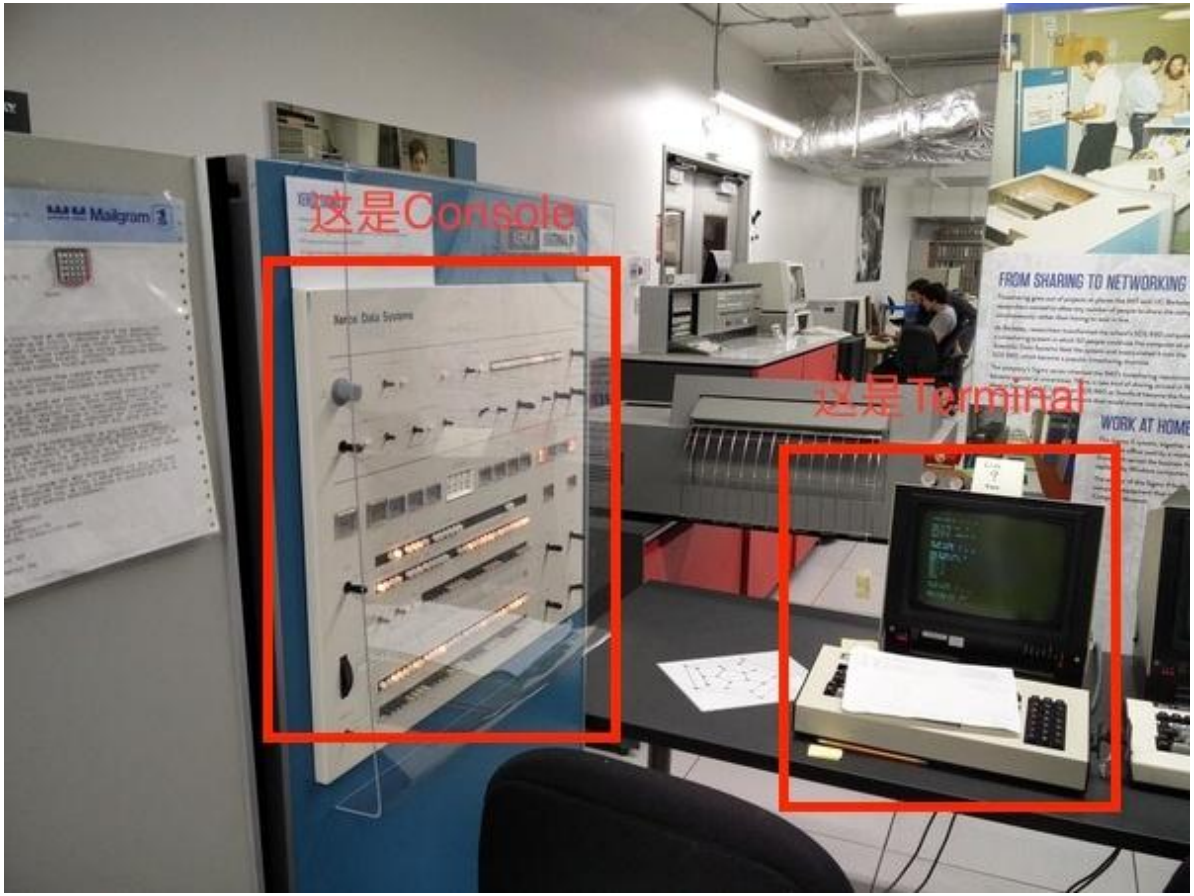
- 终端的泛化
  - 将通过串口连接的各种设备都称为终端设备
  - 串行端口终端：dev/ttySn
  - 伪终端(pseudo terminal)：/dev/pty
  - 虚拟终端(VT)：/dev/tty
  - 控制台：/dev/console

控制台的概念

- 什么是控制台？
  - 英文：console，计算机自带的输入输出设备、直接连接到计算机上
  - 一台计算机可以连接多个终端，但只能有一个控制台终端
  - 计算机启动信息、内核信息、后台服务信息会显示在控制台上
  - 终端只显示跟当前程序相关的打印信息
  - 控制台权限比终端的权限大
    - 开关机、系统设置
    - 创建用户、修改密码、权限分配
  - 控制台可以重定向到不同的终端设备上(串口、LCD、显示屏)
    - 在图形界面下，console映射到/dev/pts
    - 在命令行模式下，console映射到tty0
    - 在嵌入式环境中，console一般映射到串口、LCD上(ttyS0、ttyAMA0)

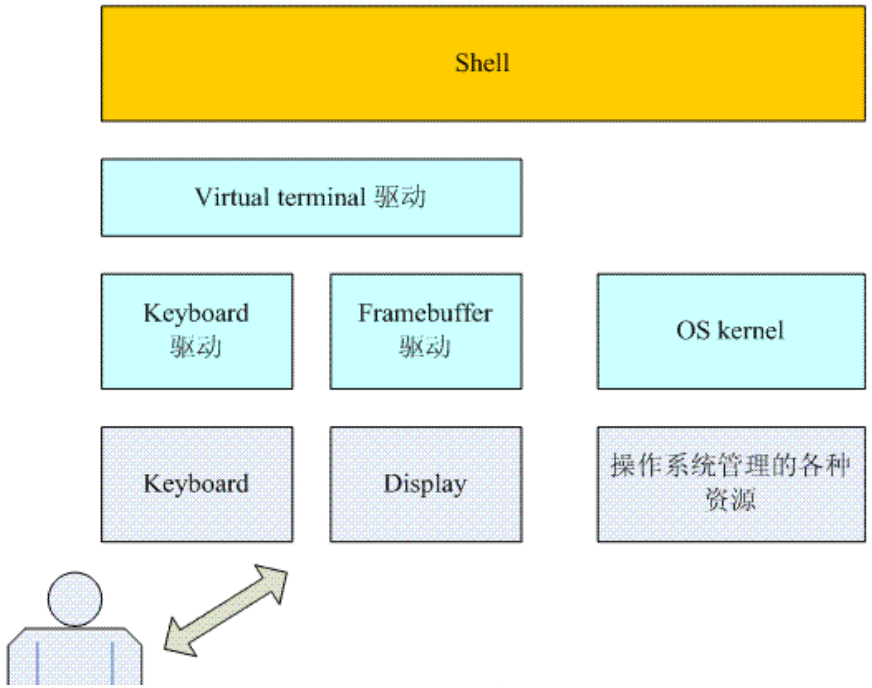
控制台的概念

终端与控制台


这是Console
这是Terminal
FROM SHARING TO NETWORKING
WORK AT HOME

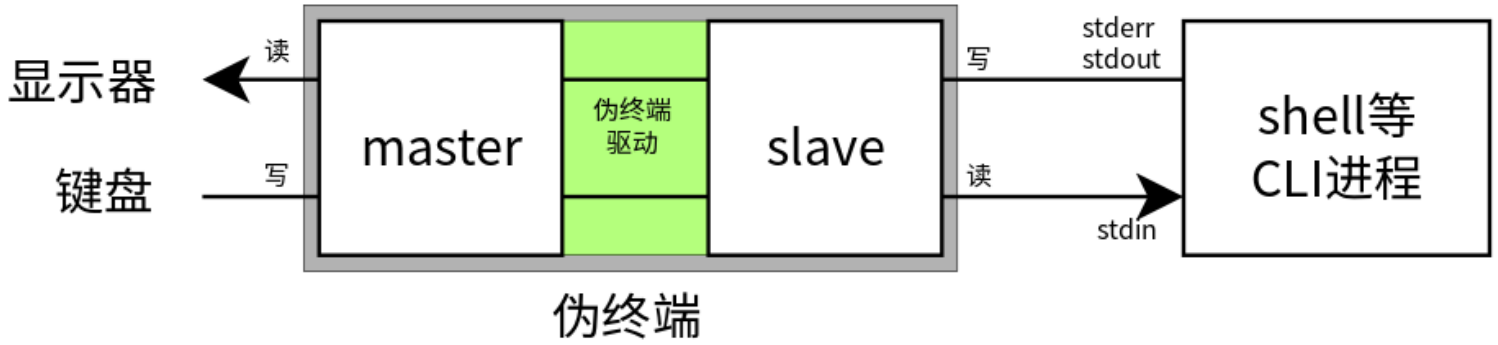
虚拟终端

- 个人PC时代
  - 终端、控制台慢慢从硬件概念演化成了软件概念
  - 虚拟终端：使用软件来模拟以前的硬件终端设备
  - Linux系统中的6虚拟终端(tty1~tty6)：使用ctrl+alt+F1~F6来回切换
  - 当前终端：当前使用的终端=控制台(/dev/console)
  - 控制终端：当前环境使用的终端：tty0



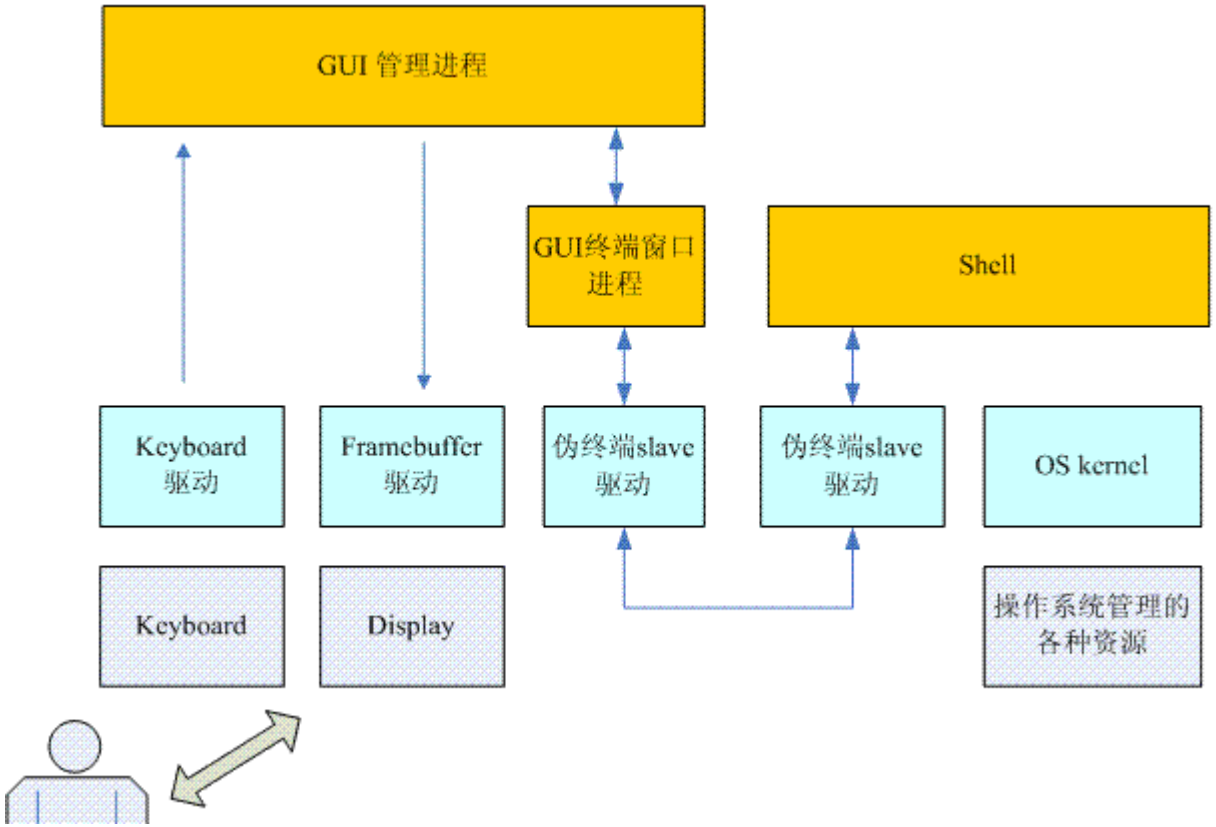
伪终端

- 伪终端的概念
  - 英文名：pseudo terminal，简称PTY
  - 用户登录(本地/SSH/telnet等)后动态创建的控制台设备文件
  - 设备文件位于/dev/pts目录下
  - Ubuntu中的Terminal其实也是一个终端模拟器
    - 为程序的输入输出提供帮助、回显、密码隐藏字符
    - 为用户提供对进程的控制：Ctrl+C 结束前台进程、发送给shell进程
    - 当用户通过SSH等软件登录主机，SSH的角色类似于terminal



伪终端

- 伪终端的软硬件架构



TIPS

- 终端与控制台的界限正越来越模糊...
- $ tty：显示当前用户所在终端的文件名

进程组与会话

进程组

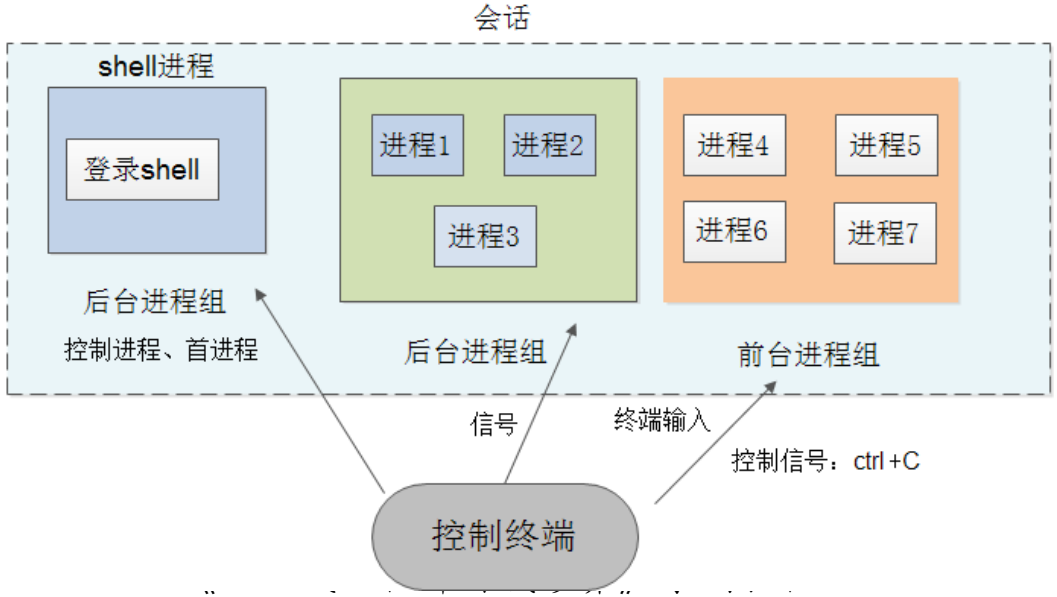
- 什么是进程组
  - 进程组：一组协同工作或关联进程的组合，每个进程组有ID(PGID)
  - 每个进程属于一个进程组，每个进程组有一个进程组长，该进程组长ID(PID)与进程组ID(PGID)相同
  - 一个信号可以发送给进程组的所有进程、让所有进程终止、暂停或继续运行

```
wit@ubuntu:/$ ps -ejH | grep "pts/17"
 5383  5383  5383 pts/17   00:00:00          bash
 5395  5395  5383 pts/17   00:00:00            su
 5396  5396  5383 pts/17   00:00:01              bash
11900 11900  5383 pts/17   00:00:05                vi
```

```
5377  5383  5383  5383 pts/17   11900 Ss    1000   0:00 |   \_ bash
5383  5395  5395  5383 pts/17   11900 S        0   0:00 |   |   \_ su
5395  5396  5396  5383 pts/17   11900 S        0   0:01 |   |       \_ bash
5396 11900 11900  5383 pts/17   11900 S+       0   0:05 |   |           \_ vi wait_v2.c
5377  6686  6686  6686 pts/18   18875 Ss    1000   0:00 |   \_ bash
6686 18875 18875  6686 pts/18   18875 R+    1000   0:00 |   |   \_ ps axjf
5377  9237  9237  9237 pts/4     9250 Ss    1000   0:00 |   \_ bash
9237  9249  9249  9237 pts/4     9250 S        0   0:00 |       \_ su
9249  9250  9250  9237 pts/4     9250 S+       0   0:02 |           \_ bash
```

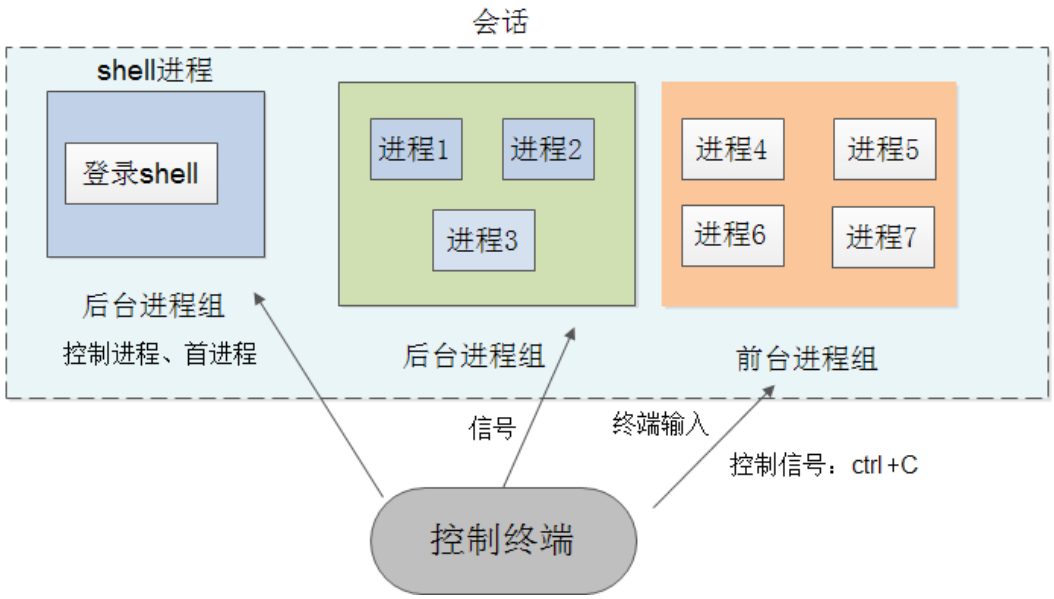
会话

- 什么是会话？
  - 会话是一个或多个进程组的集合
    - 当用户登录系统时，登录进程会为这个用户创建一个新的会话(session)
    - shell进程(如bash)作为会话的第一个进程，称为会话首进程(session leader)
    - 会话的ID(SID)：等于会话首进程的PID
    - 会话会分配给用户一个控制终端(只能有1个)，用于处理用户的输入输出
    - 一个会话包括了该登录用户的所有活动
    - 会话中的进程组由一个前台进程组和N个后台进程组构成



会话

- 进程与终端的关系
  - 控制终端：跟会话关联的终端，每个会话会分配0或1个控制终端
    - 控制进程：建立与控制终端连接的会话首进程称为控制进程
    - 终端的输入和控制信号会发送给前台进程组中的每一个进程
    - 控制终端与后台进程之间通过信号通信



会话与shell

- Shell解释器(bash)
  - 进程组和会话都是为支持shell工作而存在
    - 用户登录login、登录进程login为用户创建一个login session
    - 我们登录的这个终端设备为该会话的控制终端
    - Shell进程为该会话首进程、控制进程
    - Shell进程ID为会话的ID
  - 为了完成一项任务、shell会启动多个进程(脚本、管道命令)，这些进程会构成一个进程组
  - 会话的意义在于将很多一起协同工作或相关联的进程、进程组囊括在一个shell内，方便管理(如信号管理、资源管理等)

Shell解释器

- 什么是shell？
  - 一个读取用户命令、执行命令的程序，也被称为命令解释器
    - 用于人机交互、对shell脚本进行解释、执行
    - 内置环境变量、循环、条件语句、I/O命令、函数等
    - shell可以集成在OS内核中，也可以作为一个独立的应用进程运行
  - 登录shell进程：用户刚登录系统时，由系统创建来运行shell的进程
    - Sh：bourne shell, UNIX标配shell,支持管道、重定向、环境变量、后台执行
    - Csh：脚本语言语法与C语言类似，支持历史记录、命令行编辑等
    - Ksh：兼容sh，并吸取了csh的一些交互式特性
    - Bash：GNU项目，目前Linux上使用最广泛的shell

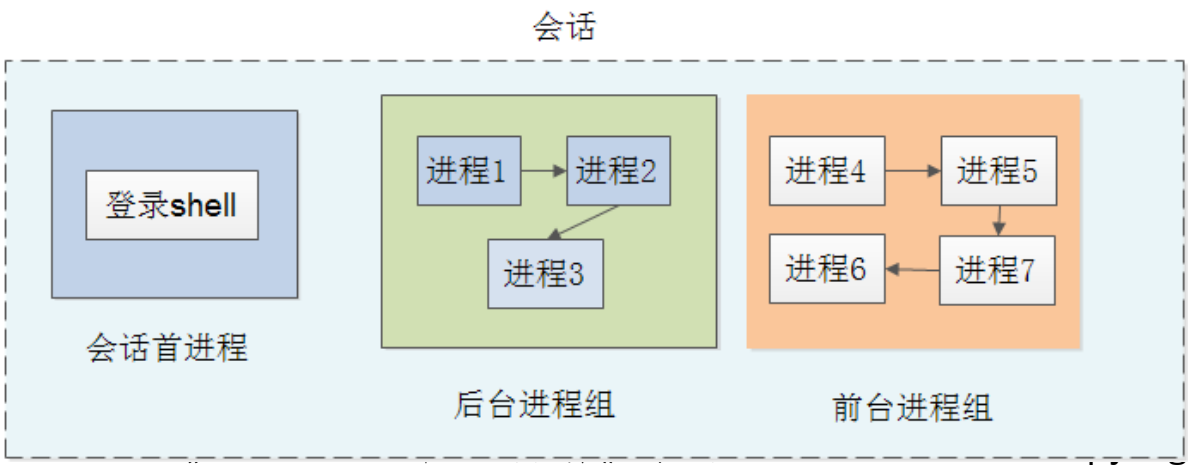
Shell的工作流程

- shell的工作流程
  - Linux内核启动后启动init进程、解析/etc/inittab 登录信息启动getty进程
  - getty进程调用setsid创建新的session和process group、提示用户登录
  - 登录后调用exec加载login程序(/bin/login)(继承getty的PID、PGID、SID)
  - Login进程对用户输入的用户名、密码进行验证
  - Login根据用户输入到/etc/passwd 查找成功后，找出对应UID、GID，并与/etc/shadow文件中对应账号的UID进行匹配验证
  - 验证成功后接着会设置该用户相关的主文件夹、启动shell交互环境
  - 验证成功后会fork子进程并通过exec执行shell(如bash\csh等)
  - 此时login和shell两个进程同属前台进程组、共享虚拟终端
  - Shell进程通过setpgid创建新的进程组、分道扬镳、并将自己设置为前台进程组，跟用户交互
  - 用户输入命令$c1 | c2 |c3，shell会fork3个子进程
  - 为这3个进程创建新的进程组，并将该进程组推向前台
  - 3个子进程执行：通过虚拟终端处理输入输出
  - 子进程执行完毕、退出，shell进程又重新回到前台，等待用户新命令

前台进程与后台进程

前台进程与后台进程

- 进程的前后台
  - 前台进程：占有控制终端的进程，其它称为后台进程
  - 后台进程：
    - shell中耗时较久的命令可以通过$ command & 后台运行
    - 好处：下一个命令不必等到上一个进程运行完才能运行
    - 一个子进程在创建时若没指定进程组，系统自动创建一个进程组、该子进程为进程组的组长，若该进程后台执行，该进程组为后台进程组
    - 会话将这些进程组囊括在一个shell终端内，选取其中一个进程组用来接收终端的输入或信号，这个进程组成为前台进程组
    - 一个会话可以有多个后台进程组，但只能有一个前台进程组



前台进程和后台进程

- Shell与前后台
  - Shell进程一开始工作在前台，等待用户输入命令
  - 用户输入命令，shell进程通过fork & exec执行命令
  - shell被提到后台，运行的命令提到前台，接受用户输入
  - 前台进程运行结束退出，shell自动被提到前台，等待用户输入
  - …

前台进程和后台进程的转换

- 进程的前后台转换
  - Shell：前台进程+任意多个后台进程
  - Ctrl + C：终止并退出前台进程，回到SHELL
  - Ctrl + Z：暂停前台命令执行，放到后台，回到SHELL
  - Jobs：查看当前在后台执行的命令
  - &：在后台执行命令
  - fg N：将进程号码为 N 的命令放到前台执行
  - bg N：将进程号码为 N 的命令放到后台执行
    - 注：该号码不是PID，是通过命令jobs看到的后台命令序号

Android中的进程

- 应用进程的淘汰机制
  - 前台进程
  - 可见进程
  - 服务进程
  - 后台进程

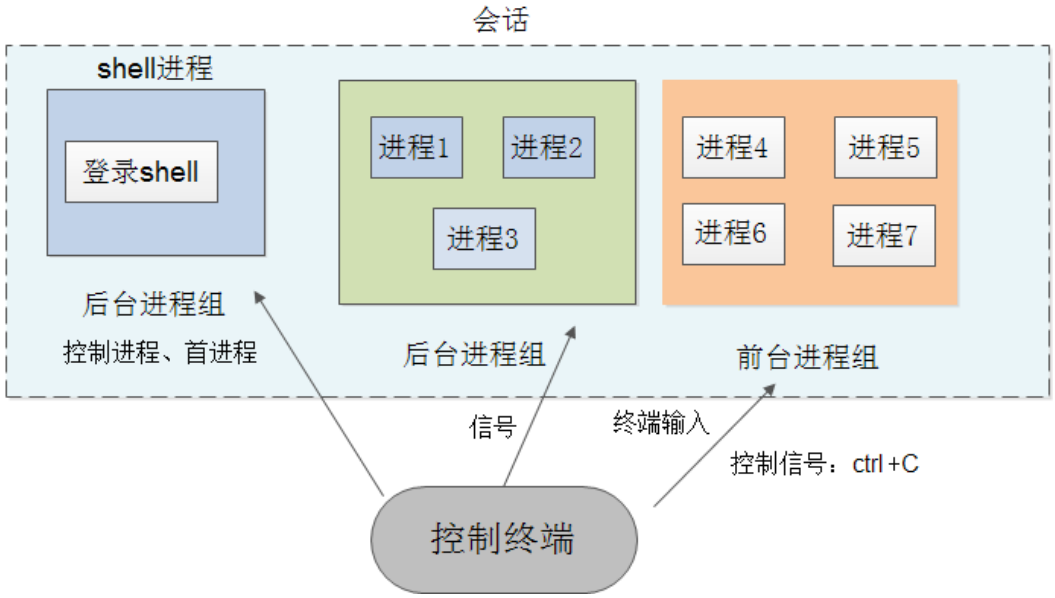
思考

- 为什么你的android手机可以实时接收微信、QQ信息？
- 为什么手机用久了会越来越卡？

守护进程

守护进程

- 守护进程的概念
  - Daemon：运行在后台的服务程序，周期性执行系统级任务或等待处理某些发生的事件(热插拔事件、信号等)
  - 独立于终端，不与任何控制终端相关联
  - 打印信息不会打印到终端上
  - 守护进程会创建自己新的会话，避免与其它会话产生关系



守护进程

- 守护进程的特点
  - 后台运行，不与控制终端相连
  - 不受用户登录或注销的影响，一直在运行，一般为系统服务进程
  - 生命周期较长，一般随系统启动和关闭，一直运行系统退出
  - 不受SIGINT、SIGQUIT、SIGTSTP跟终端相关的信号影响
  - 关闭终端不会影响daemon进程的运行
  - 守护进程命名：sshd、inetd、httpd(命名不是绝对的、通用的)

查看系统的守护进程

- 查看守护进程
  - $ ps –axj | more
    - 参数a：列出所有用户的进程
    - 参数x：不仅列出控制终端的进程，还列出所有无控制终端的进程
    - 参数j：列出与作业控制相关的信息

守护进程的应用

- Linux的守护进程
  - Linux中大多数服务器或服务程序都是使用守护进程实现的
  - Httpd：web服务器
  - Inetd：Internet服务器
  - Crond：作业规划进程
  - Syslogd：日志维护/var/log
  - Acpid：电源维护
  - Lqd：打印进程
  - Dhcpd：DHCP服务进程
  - Sshd：SSH服务进程

守护进程的启动

- 通过配置文件和脚本
  - 配置文件：/etc/
  - 启动脚本：/etc/init.d、/etc/rc*.d、/etc/rc.local
  - 启动脚本：/usr/lib/systemd/、/etc/systemd、/etc/sysconfig
  - 将普通进程“包装”成守护进程：$ nohup ./a.out &
  - $ nohup：不间断地运行命令

TIPS

- 后台进程与守护进程的区别
  - 守护进程已经完全脱离终端控制
  - 后台程序没有脱离终端控制，一些信息会输出到终端
  - 关闭终端时，后台进程会随之终止，而守护进程可以继续运行
  - 守护进程有独立的会话、文件描述符、工作目录，而后台进程是继承父进程的

编程作业

- 编写一个守护进程并运行
  - 屏蔽一些控制终端操作的信号
  - 调用fork，父进程退出
  - setsid创建一个新会话
  - 禁止进程重新打开控制终端
  - 关闭打开的文件描述符
  - 改变当前工作目录
  - 重设文件创建掩模
  - 处理SIGCHLD信号

僵尸进程

僵尸进程

- 父进程和子进程的关联
  - 在父进程中使用fork创建子进程、在调度器调度下分别调度运行
  - 子进程运行结束退出，内核释放相关资源
    - 释放占用的内存、打开的文件
    - 仍保留一定的信息：进程ID、退出状态、运行时间等
  - 父进程会调用wait/waitid 获取子进程的退出状态，释放最后的资源
- 什么是僵尸进程？
  - 如果子进程exit退出，父进程没有调用wait获取子进程状态，那么子进程的相关资源仍然保存在系统中，这种进程称为僵尸进程
  - 僵尸进程会占用PID等资源，如果系统中存在大量僵尸进程，会影响fork子进程

编程作业

- 创建一个僵尸进程，并观察进程的状态

– 进程的不同状态
  - R：task_running，可执行状态
  - S：task_interruptible，可中断的睡眠状态
  - D：task_uninterruptible，不可中断的睡眠状态
  - T：task_stopped、task_traced，暂停状态或跟踪状态
  - Z：task_dead、exit_zombie，退出状态，进程成为僵尸进程
  - X：task_dead、exit_dead，退出状态，进程即将被销毁

孤儿进程

孤儿进程

- 什么是孤儿进程？
  - 每一个进程都是从父进程fork出来的
  - 一般情况下父进程会通过wait/waitid系统调用等待子进程退出，获取到子进程状态，释放相关资源后才会退出
  - 若父进程退出时，子进程还没退出，会将进程托管给init进程，则子进程就变成了孤儿进程

编程示例

- 创建一个孤儿进程
- 观察孤儿进程的 pid 和 ppid 变化情况

思考

- 在图形模式下，孤儿进程的托管进程为upstart，pid不为1
- 在文本模式虚拟终端下，孤儿进程的托管进程init的pid为1
- 为什么？

init服务进程

- init服务进程的演变
  - Sysvinit
  - Upstart
  - Systemd

0号进程和1号进程

Linux进程的起源

- 一生二，二生三，三生万物
  - 0号进程：
    - 即idle进程，Linux内核启动后创建的第一个进程
    - 唯一没有 通过fork或者kernel_thread创建的进程
  - 1号进程：即init进程
    - 由idle进程通过kernel_thread创建，在内核空间完成初始化后，加载init程序，转变为用户空间的第一个进程
    - Linux所有用户进程都是由init进程fork创建的，init是用户进程的“祖先”
    - Init进程在系统启动后会转变为守护进程，托管孤儿进程，变为“孤儿院”
  - 2号进程
    - 即kthreadd内核线程，由kernel_thread创建，运行在内核空间
    - 负责内核线程的调度和管理

0号进程

- 从0到1
  - start_kernel：初始化内核各个组件，包括调度器，调用init_task
  - init_task，内核中所有进程、线程的task_struct的雏形
  - init_task调用kernel_thread创建内核init进程、kthreadd内核线程
  - 内核初始化后，init_task最终演变为0号进程idle
  - 内核开始调度执行，当无进程运行时，会调度idle进程运行

1号进程

- 从1到用户空间进程
  - Start_kernel->rest_init->kernel_thread(kernel_init, NULL, CLONE_FS);
  - 若用户通过init启动参数显式指定，运行用户指定的程序
  - 若没指定：kernel_init->execve(/sbin/init)运行init进程
    - /sbin/init
    - /etc/init
    - /bin/init
    - /bin/sh
  - 1号init进程从内核态转换为用户态，变为用户进程的“祖先”
  - 用户态init进程从/etc/inittab中完成各种初始化
    - 初始化系统、启动各种服务
    - 启动登录服务
    - 用户态init进程接着执行/bin/bash启动shell进程
    - 0号进程->init内核进程->1号init用户进程->getty进程->shell进程

Linux操作系统的init服务进程

- init服务进程的演变
  - sysvinit
    - 通过runlevel预定义运行模式：
      - runlevel 3为命令行模式，5为图形界面模式，0是关机，6是重启。提供各种命令：reboot、shutdown等
      - 运行位于/etc/rc*.d(一般链接到/etc/init.d)的脚本来启动各种系统服务
    - 缺点：按脚本顺序启动服务，耗时较长，不适用消费电子
  - upstart
    - 基于事件驱动机制，动态开启、关闭相关服务
    - 并行启动各种服务，启动速度快，适用于便携式设备
  - systemd
    - Linux桌面系统最新的初始化系统(init)、功能更强大
    - 采用socket与总线激活式提高各个服务的并行运行性能
    - 在Ubuntu等桌面操作系统中广泛使用

嵌入式中的init服务进程

- init服务进程的演变
  - Linuxrc：在嵌入式系统中一般指定/linuxrc为init进程
  - 设置bootargs：
    - setenv bootargs ‘mem=64M console=ttyS0,115200 root=/dev/ram rw init=/linuxrc’
  - Linuxrc：存在于根文件系统的一个应用程序
    - 在嵌入式Linux中一般是busybox，busybox是专为嵌入式开发的init应用程序，提供了一整套的shell命令集：ls、cd、ps、pwd、rm等
    - 负责系统启动前后的各种配置、引出用户界面(cmdline 或 GUI)

Linux进程全景图

Linux进程全景图

- $ pstree
  - 查看当前系统所有进程的关系
  - -a：显示系统所有的进程
  - -A：使用ASCII字符格式显示