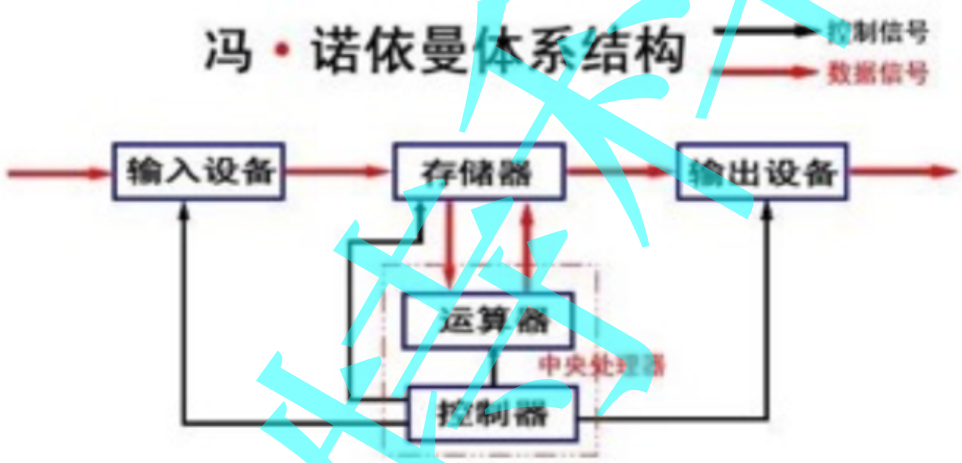
本节重点：

> - 认识冯诺依曼系统
> - 操作系统概念与定位
> - 深入理解进程概念，了解PCB
> - 学习进程状态，学会创建进程，掌握僵尸进程和孤儿进程，及其形成原因和危害
> - 了解进程调度，Linux进程优先级，理解进程竞争性与独立性，理解并行与并发
> - 理解环境变量，熟悉常见环境变量及相关指令, getenv/setenv函数
> - 理解C内存空间分配规律，了解进程内存映像和应用程序区别, 认识地址空间。
> - 选学Linux2.6 kernel，O(1)调度算法架构

冯诺依曼体系结构

我们常见的计算机，如笔记本。我们不常见的计算机，如服务器，大部分都遵守冯诺依曼体系。



截至目前，我们所认识的计算机，都是有一个个的硬件组件组成

> - 输入单元：包括键盘, 鼠标，扫描仪, 写板等
> - 中央处理器(CPU)：含有运算器和控制器等
> - 输出单元：显示器，打印机等

关于冯诺依曼，必须强调几点：

> - 这里的存储器指的是内存
> - 不考虑缓存情况，这里的CPU能且只能对内存进行读写，不能访问外设(输入或输出设备)
> - 外设(输入或输出设备)要输入或者输出数据，也只能写入内存或者从内存中读取。
> - 一句话，所有设备都只能直接和内存打交道。

对冯诺依曼的理解，不能停留在概念上，要深入到对软件数据流理解上，请解释，从你登录上qq开始和某位朋友聊天开始，数据的流动过程。从你打开窗口，开始给他发消息，到他的到消息之后的数据流动过程。如果是在qq上发送文件呢?

操作系统(Operator System)

概念

任何计算机系统都包含一个基本的程序集合，称为操作系统(OS)。笼统的理解，操作系统包括：

> - 内核（进程管理，内存管理，文件管理，驱动管理）
> - 其他程序（例如函数库，shell程序等等）

设计OS的目的

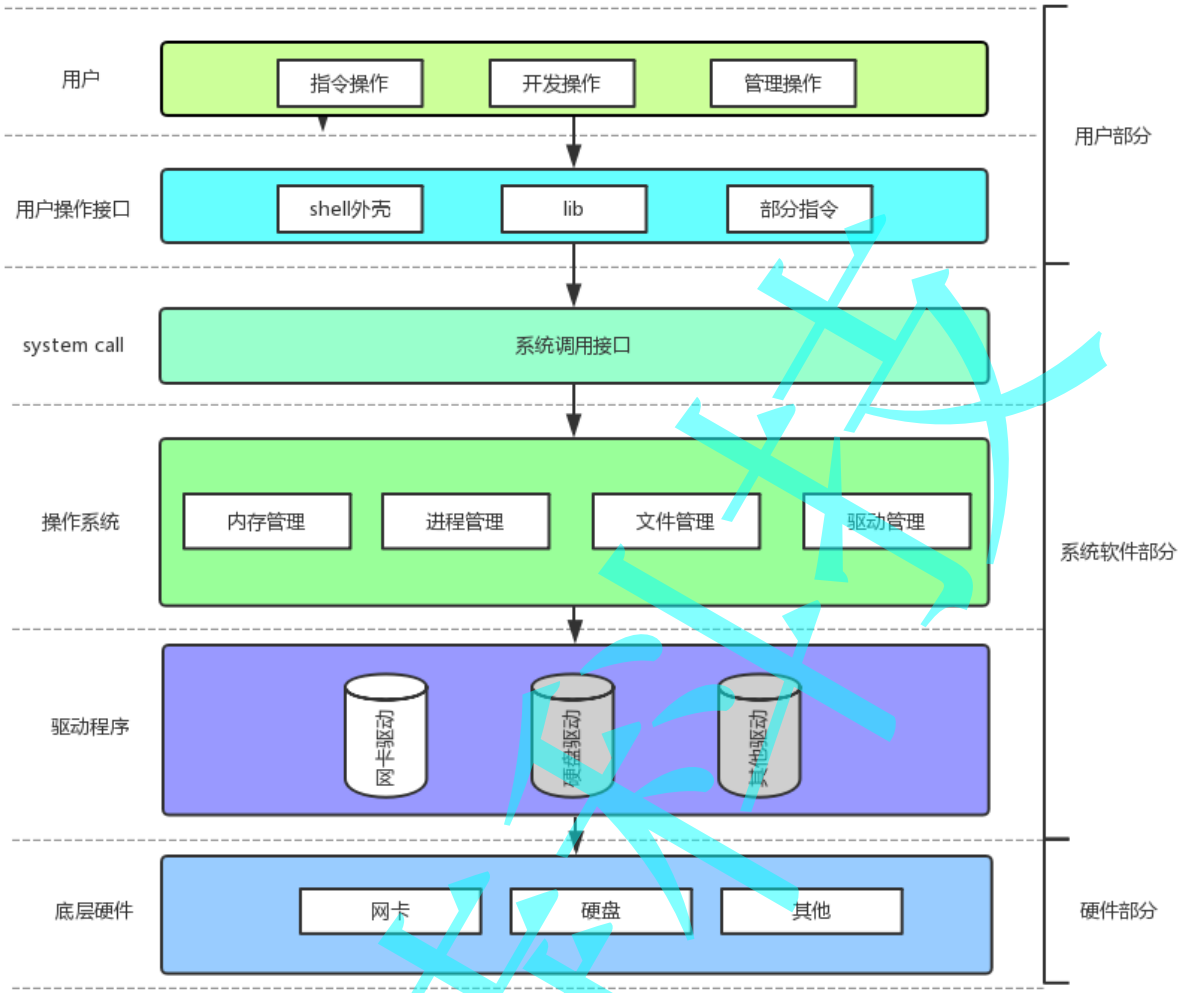
> - 与硬件交互，管理所有的软硬件资源
> - 为用户程序（应用程序）提供一个良好的执行环境

定位

> - 在整个计算机软硬件架构中，操作系统的定位是：**一款纯正的“搞管理”的软件**

如何理解 "管理"

> - 管理的例子
> - 描述被管理对象
> - 组织被管理对象



总结

计算机管理硬件

> 1. 描述起来，用struct结构体
> 2. 组织起来，用链表或其他高效的数据结构

系统调用和库函数概念

> - 在开发角度，操作系统对外会表现为一个整体，但是会暴露自己的部分接口，供上层开发使用，这部分由操作系统提供的接口，叫做系统调用。
> - 系统调用在使用上，功能比较基础，对用户的要求相对也比较高，所以，有心的开发者可以对部分系统调用进行适度封装，从而形成库，有了库，就很有利于更上层用户或者开发者进行二次开发。

承上启下

那在还没有学习进程之前，就问大家，操作系统是怎么管理进行进程管理的呢？很简单，先把进程描述起来，再把进程组织起来！

进程

基本概念

> - 课本概念：程序的一个执行实例，正在执行的程序等
> - 内核观点：担当分配系统资源（CPU时间，内存）的实体。

描述进程-PCB

> - 进程信息被放在一个叫做进程控制块的数据结构中，可以理解为进程属性的集合。
> - 课本上称之为PCB（process control block），Linux操作系统下的PCB是: task_struct

task_struct-PCB的一种

> - 在Linux中描述进程的结构体叫做task_struct。
> - task_struct是Linux内核的一种数据结构，它会被装载到RAM(内存)里并且包含着进程的信息。

task_ struct内容分类

> - 标示符: 描述本进程的唯一标示符，用来区别其他进程。
> - 状态: 任务状态，退出代码，退出信号等。
> - 优先级: 相对于其他进程的优先级。
> - 程序计数器: 程序中即将被执行的下一条指令的地址。
> - 内存指针: 包括程序代码和进程相关数据的指针，还有和其他进程共享的内存块的指针
> - **上下文数据**: 进程执行时处理器的寄存器中的数据[休学例子，要加图CPU，寄存器]。
> - I / O状态信息: 包括显示的I/O请求,分配给进程的I / O设备和被进程使用的文件列表。
> - 记账信息: 可能包括处理器时间总和，使用的时钟数总和，时间限制，记账号等。
> - 其他信息

组织进程

> 可以在内核源代码里找到它。所有运行在系统里的进程都以task_struct链表的形式存在内核里。

查看进程

进程的信息可以通过`/proc`系统文件夹查看

> - 如：要获取PID为1的进程信息，你需要查看`/proc/1`这个文件夹。

```
[root@localhost hb]# ls /proc/
1     1581  1720  1886  21    2218  2390  2448  2477  2539  2754  2856  43   809          cpuinfo      iomem        locks        partitions   sysvipc
10    16    1723  19    2102  2224  24    2450  2479  2542  2758  2890  44   9            crypto       ioports      mdstat       sched_debug  timer_list
11    160   1751  1922  2107  2278  2402  2453  2481  2554  2768  29    45   914          devices      irq          meminfo      schedstat    timer_stats
12    161   1761  2     2109  2289  2403  2456  2486  26    2771  3     5    915          diskstats    kallsyms     misc         scsi         tty
1282  1611  1762  20    2113  2295  2412  2457  2493  27    2772  30    6    965          dma          kcore        modules      self         uptime
13    162   1797  2035  2117  23    2427  2460  2498  2722  28    31    612  acpi         driver       keys         mounts       slabinfo     version
1303  1683  18    2058  2121  2347  2429  2461  2497  2728  280   32    7    asound       execdomains  key-users    mpt          softirqs     vmallocinfo
14    1695  1809  2059  2128  2357  2433  2462  25    2731  281   375   76   buddyinfo    fb           kmsg         mtd          stat         vmstat
15    17    1813  2061  2131  2365  2435  2466  2530  2733  2837  4     77   bus          filesystems  kpagecount   mtrr         swaps        zoneinfo
152   1701  1830  2074  2148  2366  2437  2469  2537  2737  2838  40    8    cgroups      fs           kpageflags   net          sys
153   1716  1865  2088  22    2386  2447  2474  2538  2739  2844  41    804  cmdline      interrupts   loadavg      pagetypeinfo sysrq-trigger
```

> - 大多数进程信息同样可以使用top和ps这些用户级工具来获取

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    while(1){
        sleep(1);
    }
    return 0;
}
```

```
[root@localhost test]# ps aux | grep test | grep -v grep
root      3239  0.0  0.0   1864   284 pts/0    S    03:40   0:00 ./test
[root@localhost test]#
```

通过系统调用获取进程标示符

- 进程id（PID）
- 父进程id（PPID）

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    printf("pid: %d\n", getpid());
    printf("ppid: %d\n", getppid());
    return 0;
}
```

通过系统调用创建进程-fork初识

- 运行`man fork`认识fork
- fork有两个返回值
- 父子进程代码共享，数据各自开辟空间，私有一份（采用写时拷贝）

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int ret = fork();
    printf("hello proc : %d!, ret: %d\n", getpid(), ret);
    sleep(1);
    return 0;
}
```

- fork 之后通常要用`if`进行分流

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int ret = fork();
    if(ret < 0){
        perror("fork");
        return 1;
    }
    else if(ret == 0){ //child
        printf("I am child : %d!, ret: %d\n", getpid(), ret);
    }else{ //father
        printf("I am father : %d!, ret: %d\n", getpid(), ret);
    }
    sleep(1);
    return 0;
}
```

进程状态 [重要点进行讲解]

看看Linux内核源代码怎么说

> - 为了弄明白正在运行的进程是什么意思，我们需要知道进程的不同状态。一个进程可以有几个状态（在Linux内核里，进程有时候也叫做任务）。 下面的状态在kernel源代码里定义：

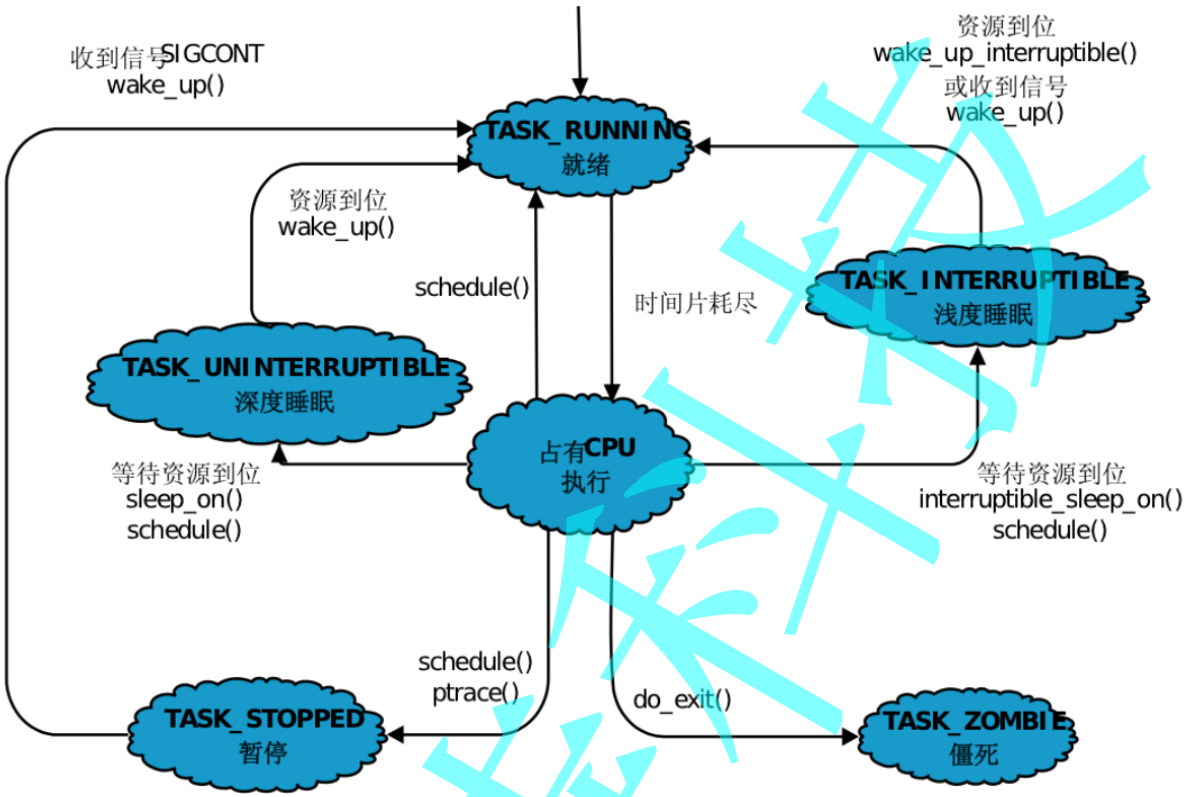
```
/*
* The task state array is a strange "bitmap" of
* reasons to sleep. Thus "running" is zero, and
* you can test for combinations of others with
* simple bit tests.
*/
static const char * const task_state_array[] = {
"R (running)", /* 0 */[重点]
"S (sleeping)", /* 1 */[重点]
"D (disk sleep)", /* 2 */
"T (stopped)", /* 4 */[重点]
"t (tracing stop)", /* 8 */
"X (dead)", /* 16 */
"Z (zombie)", /* 32 */[重点]
};
```

> - R运行状态（running）：并不意味着进程一定在运行中，它表明进程要么是在运行中要么在运行队列里。
> - S睡眠状态（sleeping): 意味着进程在等待事件完成（这里的睡眠有时候也叫做可中断睡眠（interruptible sleep））。
> - D磁盘休眠状态（Disk sleep）有时候也叫不可中断睡眠状态（uninterruptible sleep），在这个状态的进程通常会等待IO的结束。
> - T停止状态（stopped）： 可以通过发送 SIGSTOP 信号给进程来停止（T）进程。这个被暂停的进程可以通过发送 SIGCONT 信号让进程继续运行。

- X死亡状态（dead）：这个状态只是一个返回状态，你不会在任务列表里看到这个状态。

进程状态查看

```
ps aux / ps axj  命令
```

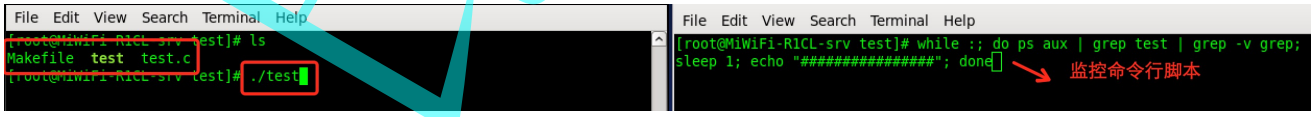


Z(zombie)-僵尸进程

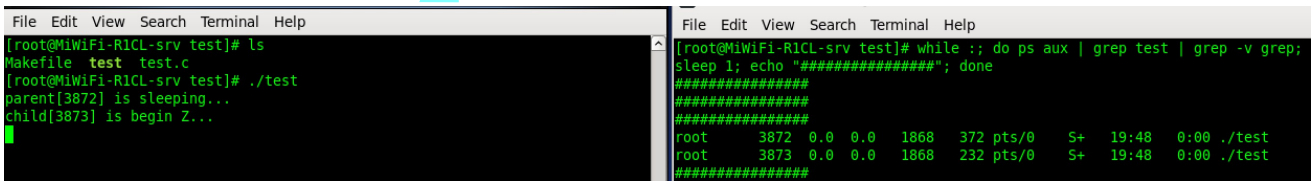
- 僵死状态（Zombies）是一个比较特殊的状态。当进程退出并且父进程（使用wait()系统调用,后面讲）没有读取到子进程退出的返回代码时就会产生僵死(尸)进程
- 僵死进程会以终止状态保持在进程表中，并且会一直在等待父进程读取退出状态代码。
- 所以，只要子进程退出，父进程还在运行，但父进程没有读取子进程状态，子进程进入Z状态

来一个创建维持30秒的僵死进程例子： #include <stdio.h> #include <stdlib.h> int main() { pid_t id = fork(); if(id < 0){ perror("fork"); return 1; } else if(id > 0){ //parent printf("parent[%d] is sleeping...\n", getpid()); sleep(30); }else{ printf("child[%d] is begin Z...\n", getpid()); sleep(5); exit(EXIT_SUCCESS); } return 0; }

编译并在另一个终端下启动监控



开始测试



看到结果

```
#################
root      3872  0.0  0.0   1868   372 pts/0    S+   19:48   0:00 ./test
root      3873  0.0  0.0      0     0 pts/0    Z+   19:48   0:00 [test] <def
unct>
```

ptrace系统调用追踪进程运行，有兴趣研究一下

僵尸进程危害

> - 进程的退出状态必须被维持下去，因为他要告诉关心它的进程（父进程），你交给我的任务，我办的怎么样了。可父进程如果一直不读取，那子进程就一直处于Z状态？是的！
> - 维护退出状态本身就是要用数据维护，也属于进程基本信息，所以保存在task_struct(PCB)中，换句话说，Z状态一直不退出，PCB一直都要维护？是的！
> - 那一个父进程创建了很多子进程，就是不回收，是不是就会造成内存资源的浪费？是的！因为数据结构对象本身就要占用内存，想想C中定义一个结构体变量（对象），是要在内存的某个位置进行开辟空间！
> - 内存泄漏?是的！
> - 如何避免？后面讲

进程状态总结

> - 至此，值得关注的进程状态全部讲解完成，下面来认识另一种进程

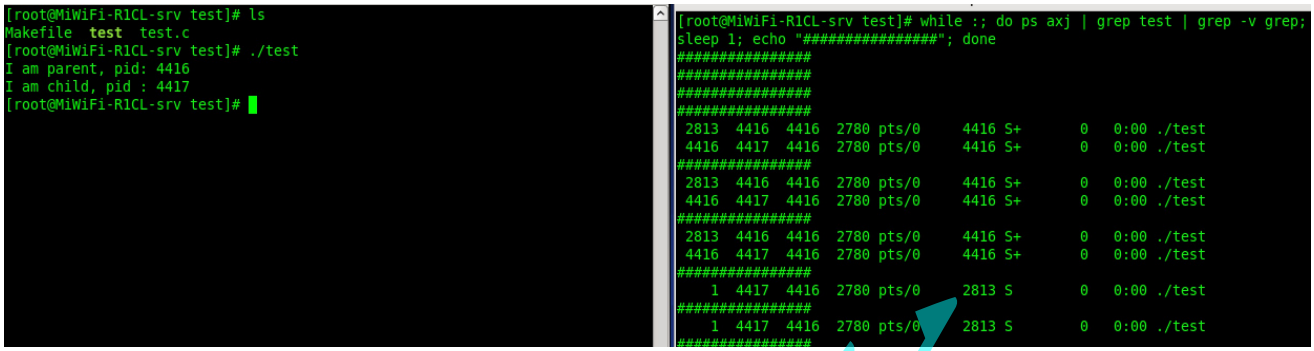
孤儿进程

> - 父进程如果提前退出，那么子进程后退出，进入Z之后，那该如何处理呢？
> - 父进程先退出，子进程就称之为“孤儿进程”
> - 孤儿进程被1号init进程领养，当然要有init进程回收喽。

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    pid_t id = fork();
    if(id < 0){
        perror("fork");
        return 1;
    }
    else if(id == 0){//child
        printf("I am child, pid : %d\n", getpid());
        sleep(10);
    }else{//parent
        printf("I am parent, pid: %d\n", getpid());
        sleep(3);
        exit(0);
    }
    return 0;
}
```

来段代码：



环境变量［重要点进行讲解］

基本概念

- 环境变量(environment variables)一般是指在操作系统中用来指定操作系统运行环境的一些参数
- 如：我们在编写C/C++代码的时候，在链接的时候，从来不知道我们的所链接的动态静态库在哪里，但是照样可以链接成功，生成可执行程序，原因就是有相关环境变量帮助编译器进行查找。
- 环境变量通常具有某些特殊用途，还有在系统当中通常具有全局特性

常见环境变量

- PATH : 指定命令的搜索路径 ［重点］
- HOME : 指定用户的主工作目录(即用户登陆到Linux系统中时,默认的目录)［重点］
- SHELL : 当前Shell,它的值通常是/bin/bash。

查看环境变量方法

`echo $NAME` //NAME:你的环境变量名称［重点］

测试PATH［重点］

1. 创建hello.c文件

```
#include <stdio.h>

int main()
{
    printf("hello world!\n");
    return 0;
}
```

2. 对比./hello执行和之间hello执行
3. 为什么有些指令可以直接执行，不需要带路径，而我们的二进制程序需要带路径才能执行?
4. 将我们的程序所在路径加入环境变量PATH当中,`export PATH=$PATH:hello程序所在路径`
5. 对比测试
6. 还有什么方法可以不用带路径，直接就可以运行呢?

测试HOME

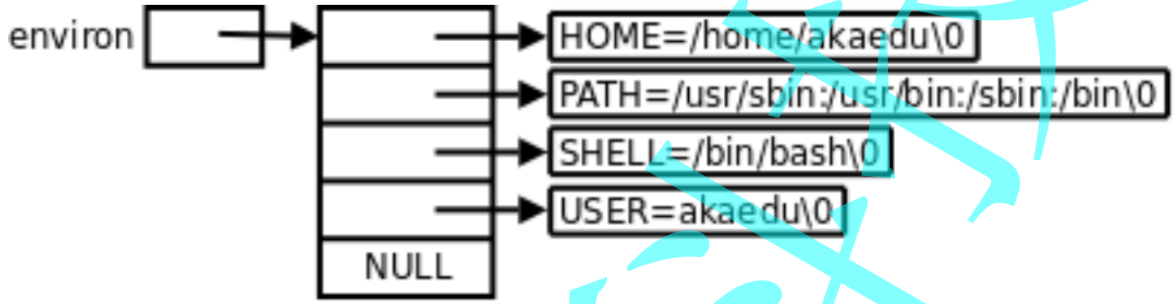
1. 用root和普通用户，分别执行`echo $HOME`,对比差异

> 2. 执行`cd ~; pwd`,对应`~`和`HOME`的关系

和环境变量相关的命令

> 1. echo: 显示某个环境变量值［重点］
> 2. export: 设置一个新的环境变量［重点］
> 3. env: 显示所有环境变量［重点］
> 4. unset: 清除环境变量
> 5. set: 显示本地定义的shell变量和环境变量

环境变量的组织方式



每个程序都会收到一张环境表，环境表是一个字符指针数组，每个指针指向一个以'\0'结尾的环境字符串

通过代码如何获取环境变量

> - 命令行第三个参数［重点］

```
#include <stdio.h>

int main(int argc, char *argv[], char *env[])
{
    int i = 0;
    for(; env[i]; i++){
        printf("%s\n", env[i]);
    }
    return 0;
}
```

> - 通过第三方变量environ获取［重点］

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    extern char **environ;
    int i = 0;
    for(; environ[i]; i++){
        printf("%s\n", environ[i]);
    }
    return 0;
}
```

libc中定义的全局变量environ指向环境变量表,environ没有包含在任何头文件中,所以在使用时 要用extern声明。

通过系统调用获取或设置环境变量

> - `putenv`, 后面讲解
> - `getenv`, 本次讲解

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("%s\n", getenv("PATH"));
    return 0;
}
```

常用getenv和putenv函数来访问特定的环境变量。

环境变量通常是具有全局属性的［重点］

> - 环境变量通常具有全局属性，可以被子进程继承下去

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char * env = getenv("MYENV");
    if(env){
        printf("%s\n", env);
    }
    return 0;
}
```

直接查看，发现没有结果，说明该环境变量根本不存在

> - 导出环境变量 `export MYENV="hello world"`
> - 再次运行程序，发现结果有了！说明：环境变量是可以被子进程继承下去的！想想为什么？

实验

> - 如果只进行`MYENV=“helloworld”`,不调用export导出，在用我们的程序查看，会有什么结果？为什么?
> - 普通变量

程序地址空间

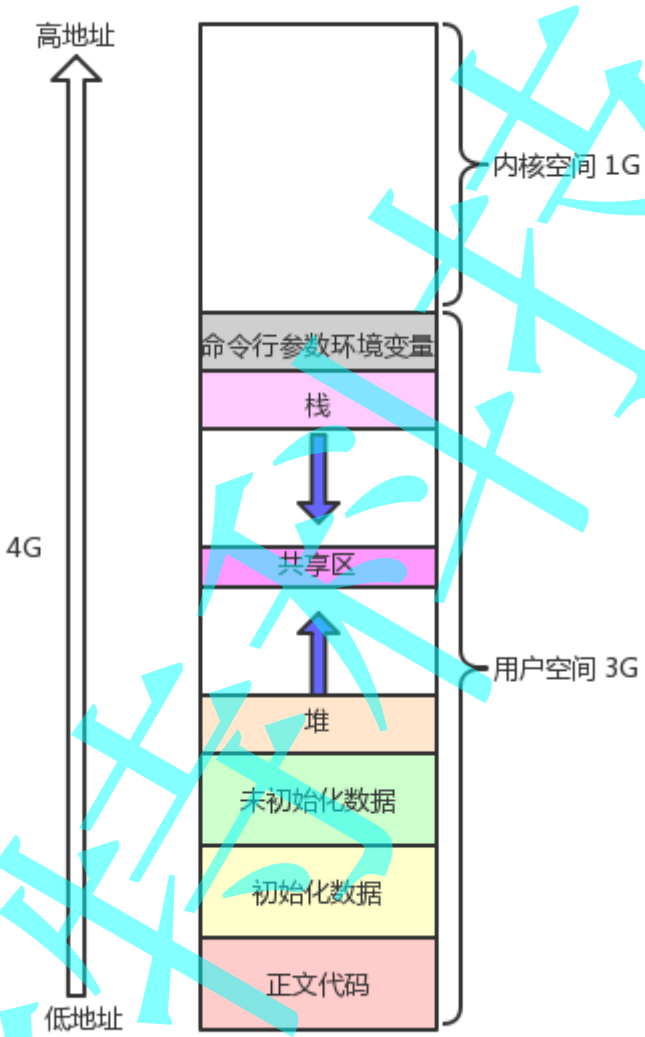
研究背景

> - kernel 2.6.32

- 32位平台

程序地址空间回顾

我们在讲C语言的时候，老师给大家画过这样的空间布局图



可是我们对他并不理解！

来段代码感受一下

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int g_val = 0;

int main()
{
    pid_t id = fork();
```

```
    if(id < 0){
        perror("fork");
        return 0;
    }
    else if(id == 0){ //child
        printf("child[%d]: %d : %p\n", getpid(), g_val, &g_val);
    }else{ //parent
        printf("parent[%d]: %d : %p\n", getpid(), g_val, &g_val);
    }
    sleep(1);
    return 0;
}
```

输出

```
//与环境相关，观察现象即可
parent[2995]: 0 : 0x80497d8
child[2996]: 0 : 0x80497d8
```

我们发现，输出出来的变量值和地址是一模一样的，很好理解呀，因为子进程按照父进程为模版，父子并没有对变量进行进行任何修改。可是将代码稍加改动:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int g_val = 0;

int main()
{
    pid_t id = fork();
    if(id < 0){
        perror("fork");
        return 0;
    }
    else if(id == 0){ //child,子进程肯定先跑完，也就是子进程先修改，完成之后，父进程再读取
        g_val=100;
        printf("child[%d]: %d : %p\n", getpid(), g_val, &g_val);
    }else{ //parent
        sleep(3);
        printf("parent[%d]: %d : %p\n", getpid(), g_val, &g_val);
    }
    sleep(1);
    return 0;
}
```

输出结果:

```
//与环境相关，观察现象即可
child[3046]: 100 : 0x80497e8
parent[3045]: 0 : 0x80497e8
```

我们发现，父子进程，输出地址是一致的，但是变量内容不一样！能得出如下结论:

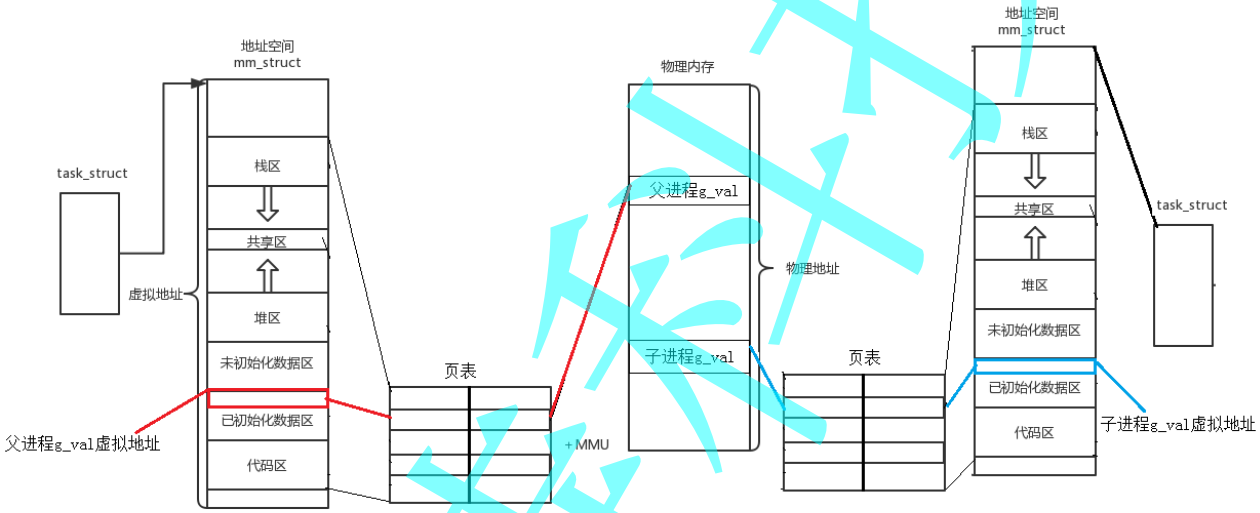
> - 变量内容不一样,所以父子进程输出的变量绝对不是同一个变量
> - 但地址值是一样的，说明，该地址绝对不是物理地址！
> - 在Linux地址下，这种地址叫做`虚拟地址`
> - 我们在用C/C++语言所看到的地址，全部都是虚拟地址！物理地址，用户一概看不到，由OS统一管理

OS必须负责将`虚拟地址`转化成`物理地址`。

进程地址空间

所以之前说'程序的地址空间'是不准确的，准确的应该说成`进程地址空间`，那该如何理解呢？看图：

分页&虚拟地址空间



说明:

> - 上面的图就足矣说名问题，同一个变量，地址相同，其实是虚拟地址相同，内容不同其实是被映射到了不同的物理地址！

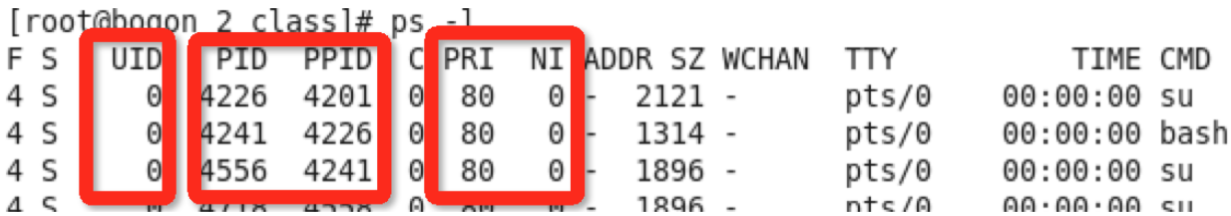
进程优先级［选学］

基本概念

> - cpu资源分配的先后顺序，就是指进程的优先权（priority）。
> - 优先权高的进程有优先执行权利。配置进程优先权对多任务环境的linux很有用，可以改善系统性能。
> - 还可以把进程运行到指定的CPU上，这样一来，把不重要的进程安排到某个CPU，可以大大改善系统整体性能。

查看系统进程

在linux或者unix系统中，用ps –l命令则会类似输出以下几个内容:



我们很容易注意到其中的几个重要信息，有下：

- UID : 代表执行者的身份
- PID : 代表这个进程的代号
- PPID： 代表这个进程是由哪个进程发展衍生而来的，亦即父进程的代号
- PRI： 代表这个进程可被执行的优先级，其值越小越早被执行
- NI： 代表这个进程的nice值

PRI and NI

- PRI也还是比较好理解的，即进程的优先级，或者通俗点说就是程序被CPU执行的先后顺序，此值越小进程的优先级别越高
- 那NI呢?就是我们所要说的nice值了，其表示进程可被执行的优先级的修正数值
- PRI值越小越快被执行，那么加入nice值后，将会使得PRI变为：PRI(new)=PRI(old)+nice
- 这样，当nice值为负值的时候，那么该程序将会优先级值将变小，即其优先级会变高，则其越快被执行
- 所以，调整进程优先级，在Linux下，就是调整进程nice值
- nice其取值范围是-20至19，一共40个级别。

PRI vs NI

- 需要强调一点的是，进程的nice值不是进程的优先级，他们不是一个概念，但是进程nice值会影响到进程的优先级变化。
- 可以理解nice值是进程优先级的修正修正数据

查看进程优先级的命令

用top命令更改已存在进程的nice：

- top
- 进入top后按“r”->输入进程PID->输入nice值

其他概念

- 竞争性: 系统进程数目众多，而CPU资源只有少量，甚至1个，所以进程之间是具有竞争属性的。为了高效完成任务，更合理竞争相关资源，便具有了优先级
- 独立性: 多进程运行，需要独享各种资源，多进程运行期间互不干扰
- 并行: 多个进程在多个CPU下分别，同时进行运行，这称之为并行
- 并发: 多个进程在一个CPU下采用进程切换的方式，在一段时间之内，让多个进程都得以推进，称之为并发