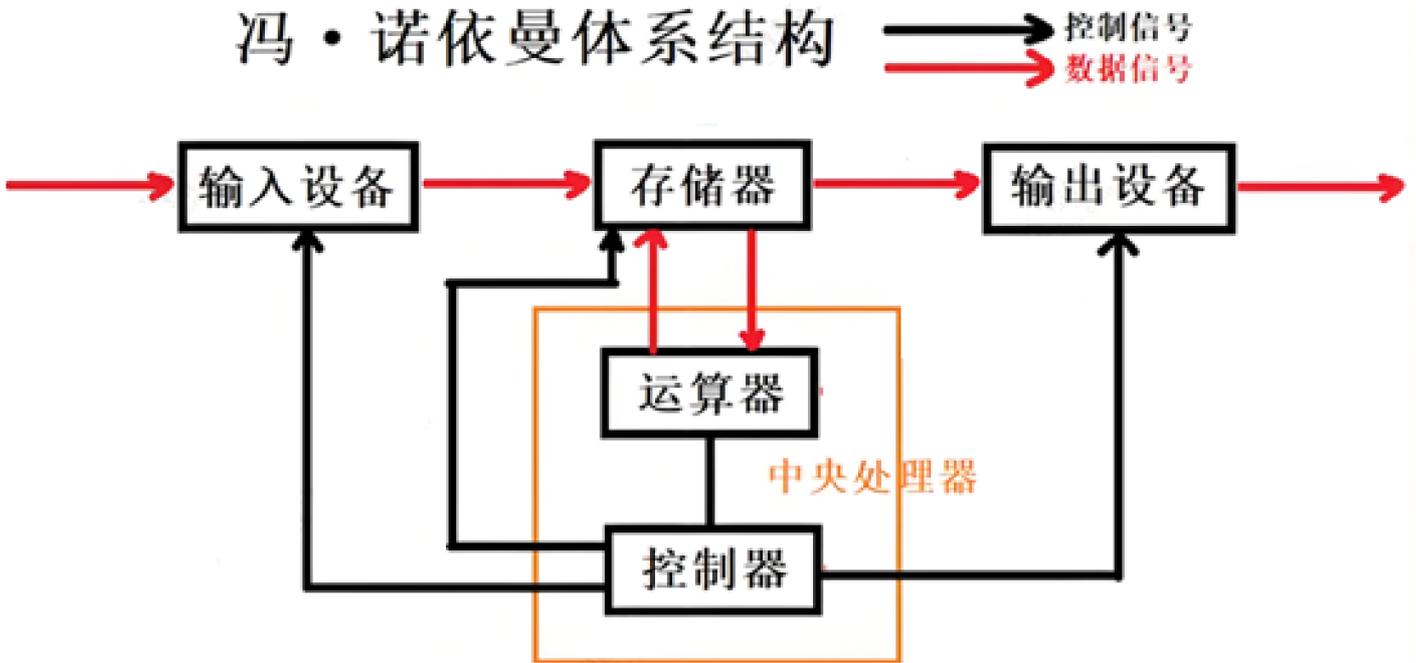


# 进程

## 1. 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

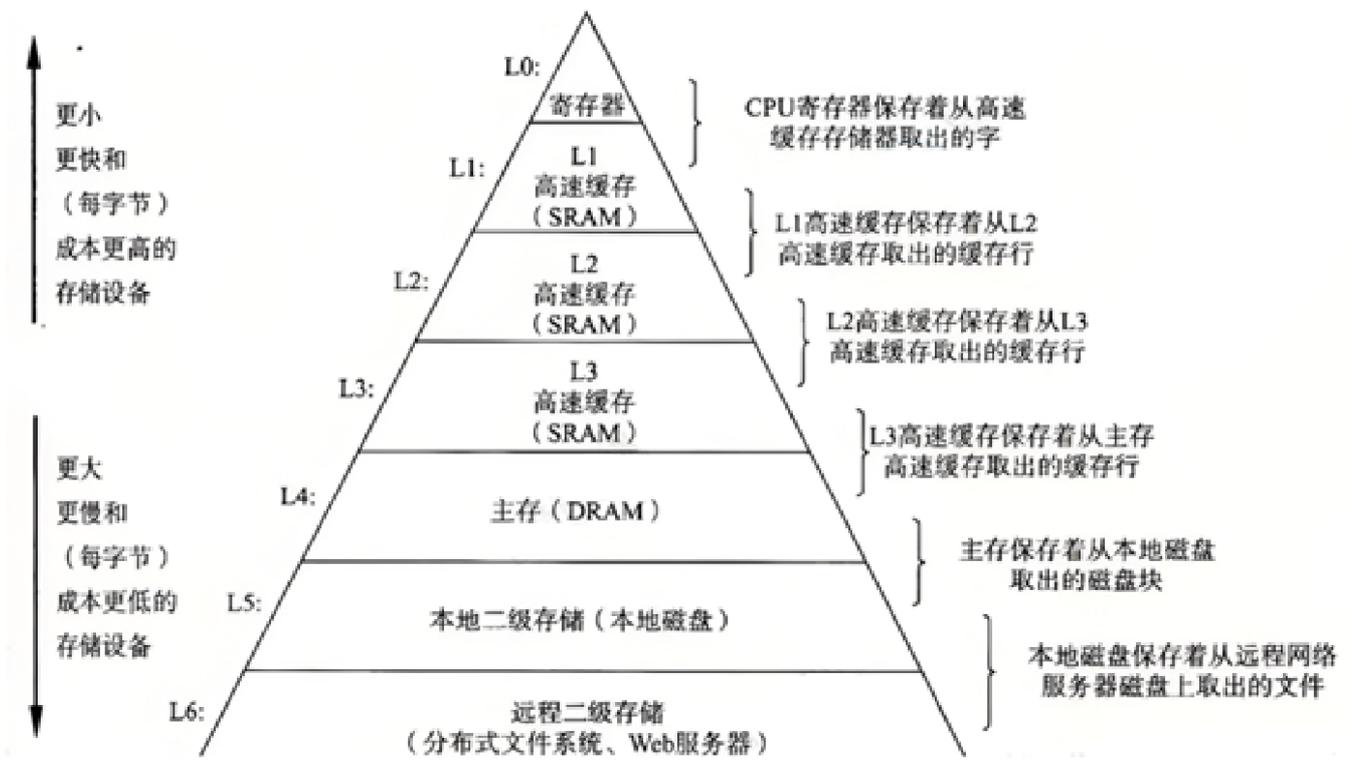


截至目前，我们所认识的计算机，都是由一个个的硬件组件组成

- 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：**

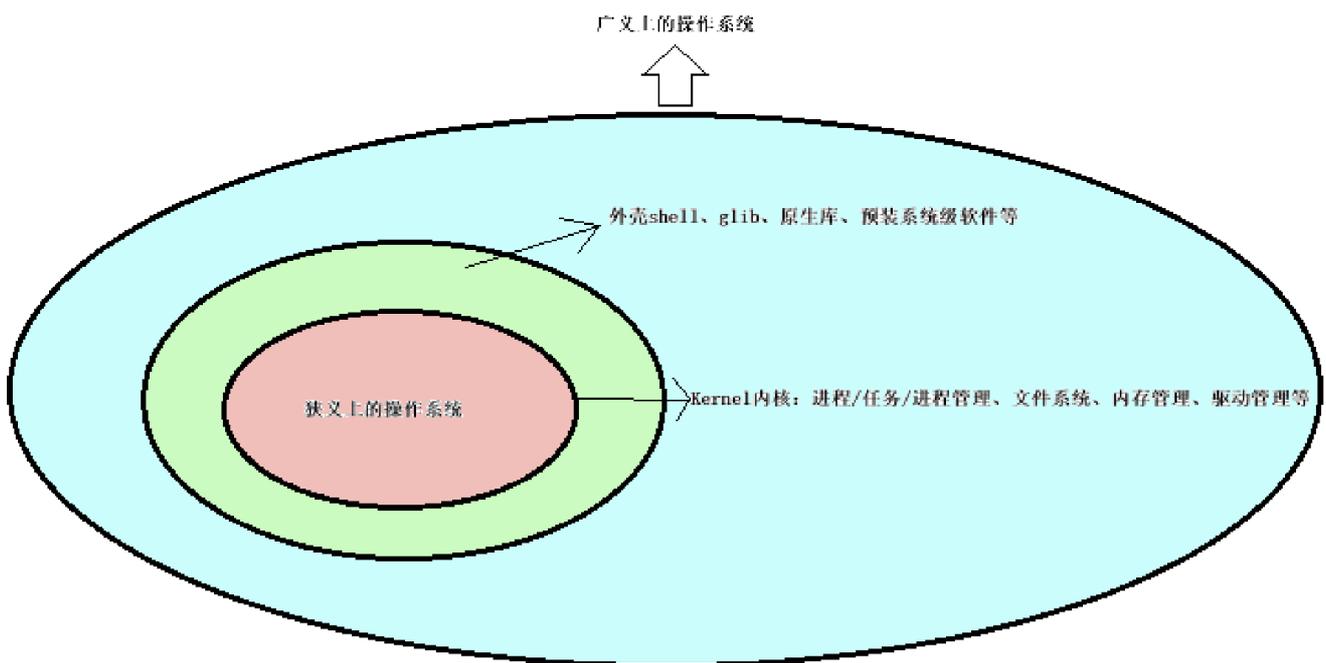
- 这里的存储器指的是内存
- 不考虑缓存情况，这里的CPU能且只能对内存进行读写，不能访问外设(输入或输出设备)(数据层面)
- 外设(输入或输出设备)要输入或者输出数据，也只能写入内存或者从内存中读取。
- 一句话，所有设备都只能直接和内存打交道。

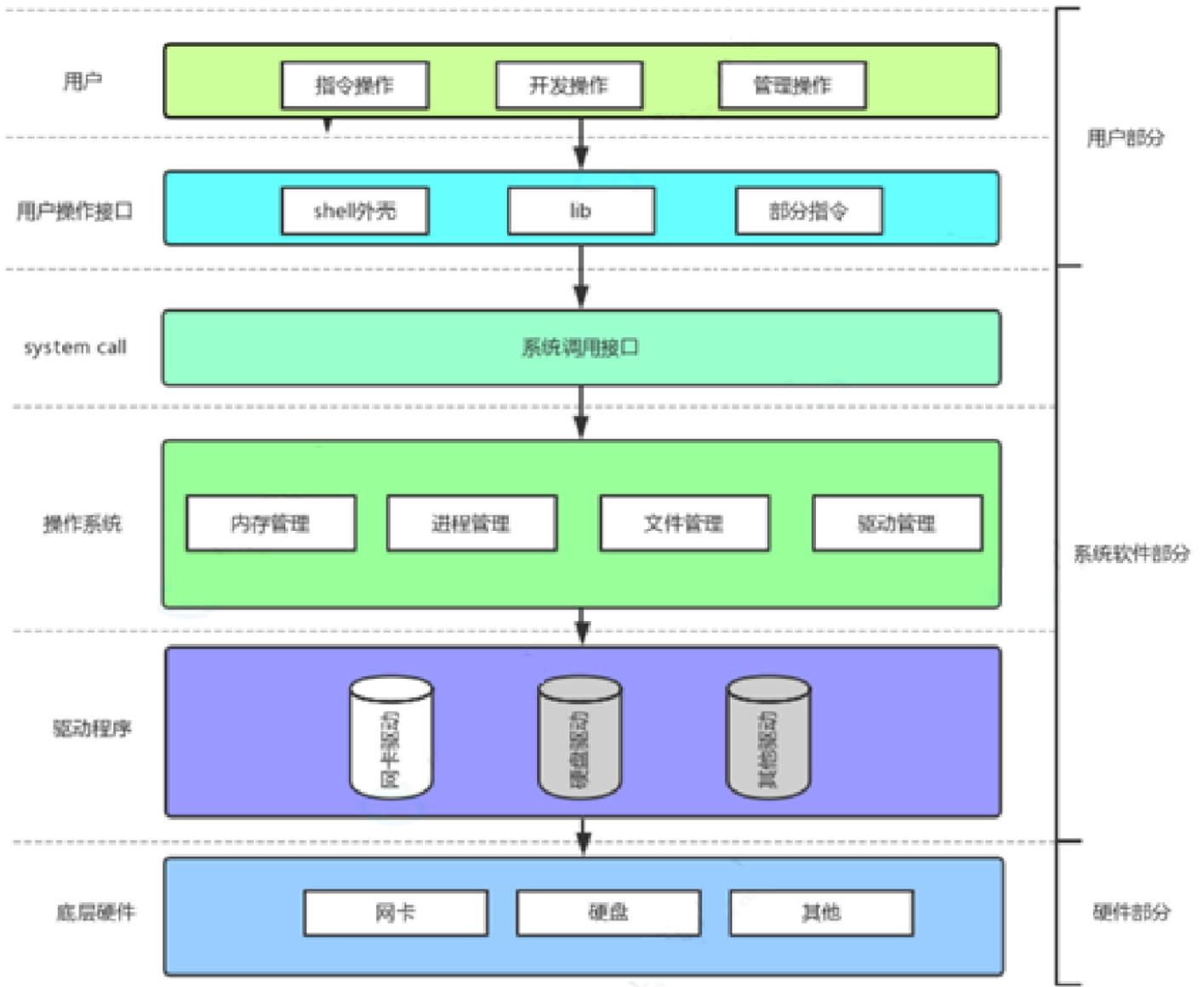


## 2. 操作系统(Operator System)

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）





### 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。

## 3. 进程

### 3-1 基本概念与基本操作

- 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。
- 当前：进程 = 内核数据结构(task\_struct) + 自己的程序代码和数据

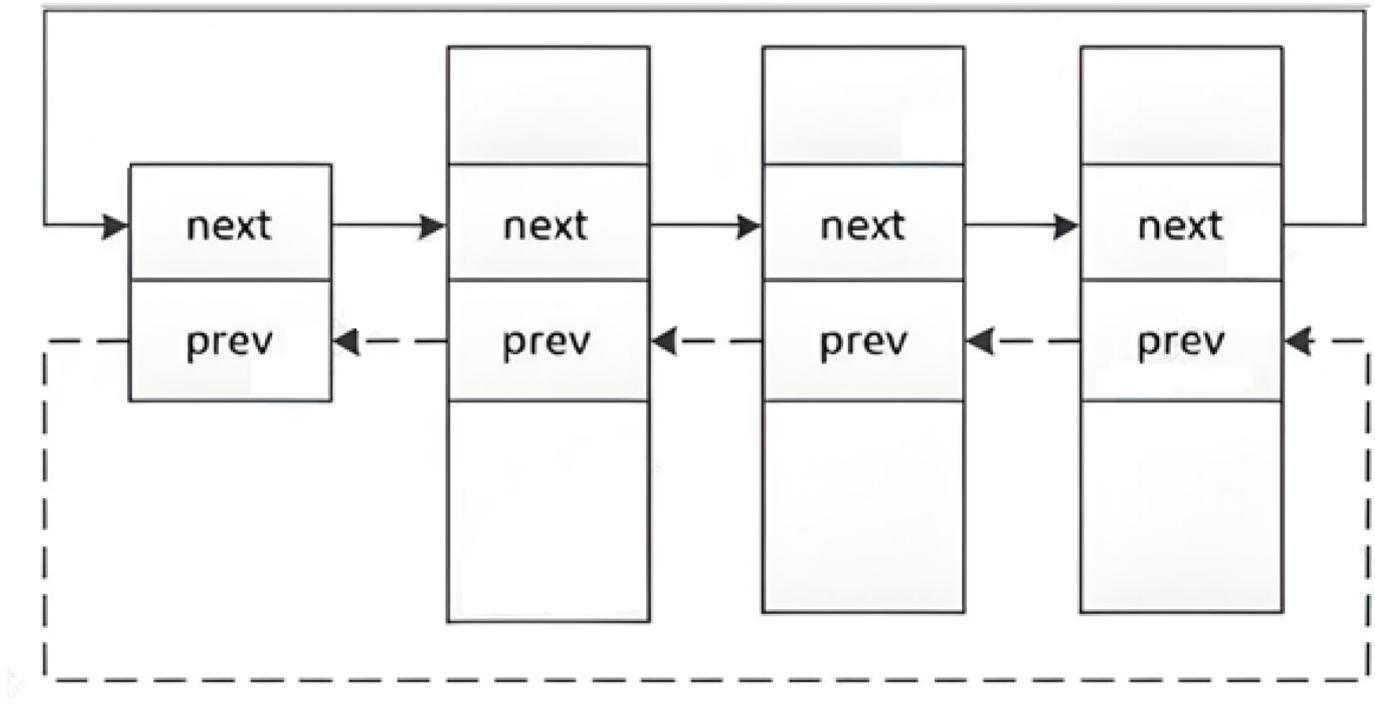
### 3-2 task\_struct

#### 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
- 具体详细信息后续会介绍

#### 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 双链表的形式存在内核里



### 3-1-6 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）

代码块

```

1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  int main()
5  {
6      int ret = fork();
7      if (ret < 0)
8      {
9          perror("fork");
10         return 1;
11     }
12
13     else if (ret == 0)
14     { // child
15         printf("I am child : %d!, ret: %d\n", getpid(), ret);
16     }
17
18     else
19     { // father
20         printf("I am father : %d!, ret: %d\n", getpid(), ret);

```

```
21     }
22
23     sleep(1);
24     return 0;
25 }
```



- fork为什么会有两个返回值?
- 两个返回值各种给父子如何返回?
- 至于：一个变量怎么能让 `if` 和 `else if` 同时成立这个问题，需要在后面才能解释清楚。

我来总结，你看对不对。其实实际上就是fork让父进程生成一个子进程，其中子进程继承父进程的所有内容，所以也会同时执行fork，所以会有两个返回值，而且也都有各自的pid。之所以if与else if同时成立，是因为根据fork的返回值进入不同的进程

你的总结**完全正确**！👍 让我再清晰地梳理一下：

## 你的理解完全正确：

### 1. 两个返回值的原因：

- `fork()` 创建子进程后，子进程会复制父进程的所有内容，包括执行到的代码位置
- 所以父子进程都会从 `fork()` 调用之后继续执行，各自获得一个返回值

### 2. 返回值分配：

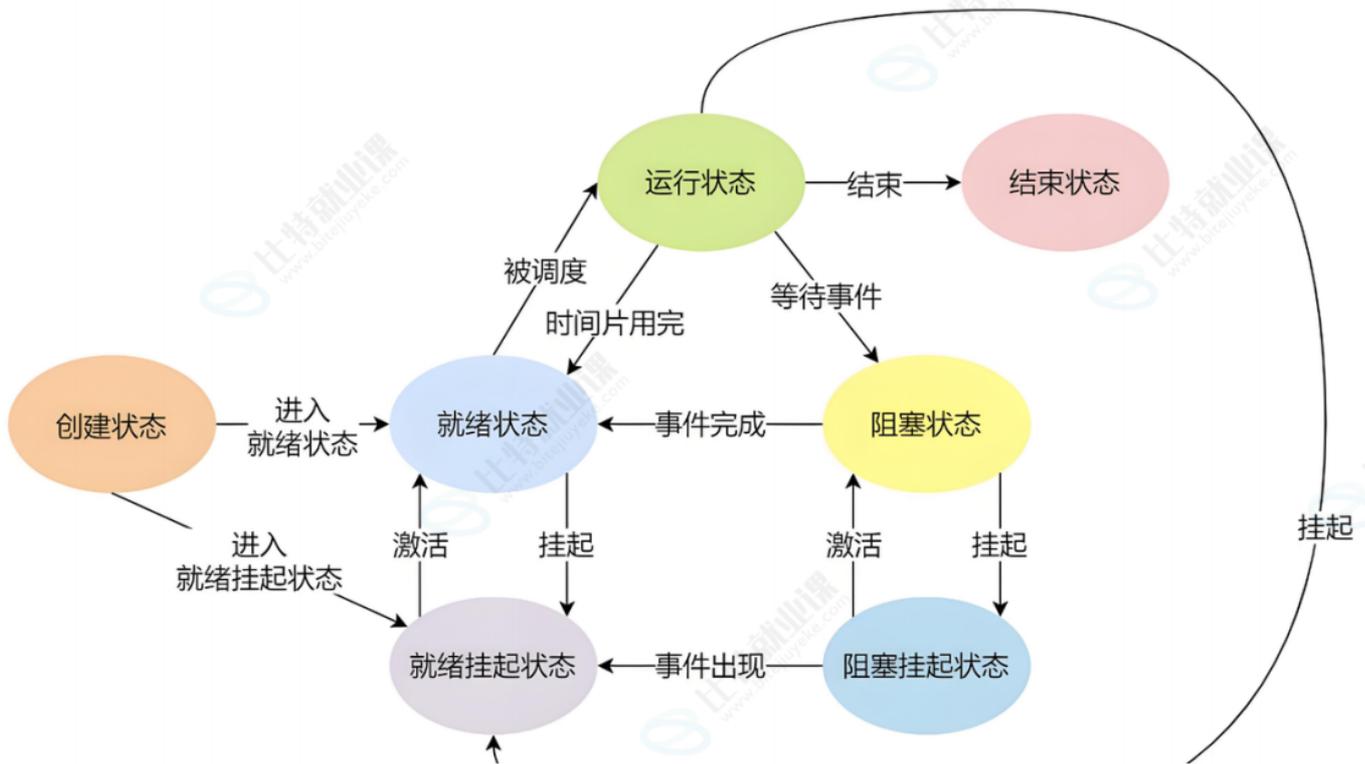
- 父进程得到子进程的PID（大于0）
- 子进程得到0
- 两个进程都有自己独立的 `getpid()` 值

### 3. if-else if "同时成立"的真相：

- 实际上**不是在一个进程中同时成立**
- 而是**两个不同的进程**分别进入了不同的分支：
  - 父进程进入 `else` 分支
  - 子进程进入 `else if (ret == 0)` 分支
- 由于输出几乎同时出现，给用户造成了"同时成立"的错觉



## 3-2 进程状态



### 3-2-1 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：

代码块

```

1  /*
2   *The task state array is a strange "bitmap" of
3   *reasons to sleep. Thus "running" is zero, and
4   *you can test for combinations of others with
5   *simple bit tests.
6   */
7  static const char *const task_state_array[] = {
8      "R (running)",    /*0 */
9      "S (sleeping)",  /*1 */
10     "D (disk sleep)", /*2 */
11     "T (stopped)",    /*4 */
12     "t (tracing stop)", /*8 */
13     "X (dead)",       /*16 */
14     "Z (zombie)",     /*32 */
15 };

```

· 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。

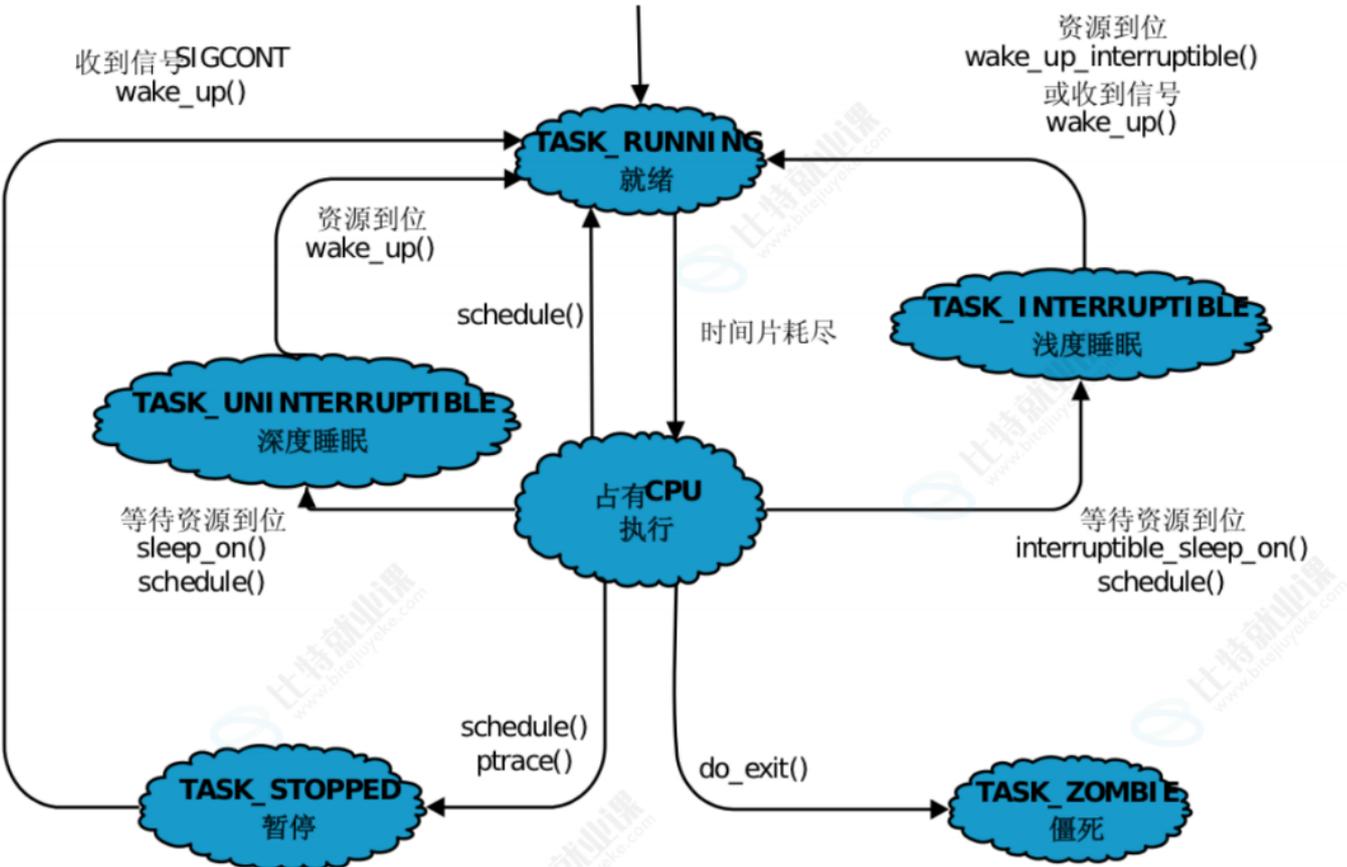
- S睡眠状态 (sleeping): 意味着进程在等待事件完成 (这里的睡眠有时候也叫做可中断睡眠 (interruptible sleep) ) 。
- D磁盘休眠状态 (Disk sleep) 有时候也叫不可中断睡眠状态 (uninterruptible sleep) ，在这个状态的进程通常会等待IO的结束。
- T停止状态 (stopped) ： 可以通过发送 SIGSTOP 信号给进程来停止 (T) 进程。这个被暂停的进程可以通过发送 SIGCONT 信号让进程继续运行。
- X死亡状态 (dead) ： 这个状态只是一个返回状态，你不会在任务列表里看到这个状态。

### 3-2-2 进程状态查看

代码块

```
1 ps aux / ps axj 命令
```

- a: 显示一个终端所有的进程，包括其他用户的进程。
- x: 显示没有控制终端的进程，例如后台运行的守护进程。
- j: 显示进程归属的进程组ID、会话ID、父进程ID，以及与作业控制相关的信息
- u: 以用户为中心的格式显示进程信息，提供进程的详细信息，如用户、CPU和内存使用情况等



### 3-2-3 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

### 3-2-4 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!

### 3-2-5 孤儿进程

- 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/systemd进程领养, 当然要有init/systemd进程回收喽。

代码块

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  int main()
5  {
6      pid_t id = fork();
7      if (id < 0)
8      {
9          perror("fork");
10         return 1;
11     }
12
13     else if (id == 0)
14     { // child
```

```

15     printf("I am child, pid : %d\n", getpid());
16     sleep(10);
17 }
18
19 else
20 { // parent
21     printf("I am parent, pid: %d\n", getpid());
22     sleep(3);
23     exit(0);
24 }
25
26     return 0;
27 }
28

```

## 3-3 进程优先级

### 3-3-1 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，可以大改善系统整体性能。

### 3-3-2 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：

```

[whb@bite-alicloud processbar]$ ps -l
F S  UID  PID  PPID  C  PRI  NI  ADDR  SZ  WCHAN  TTY  TIME  CMD
0 R  1000 1257 24278 0  80  0  - 38328  -  pts/0  00:00:00 ps
0 S  1000 24278 24277 0  80  0  - 28919  do_wai pts/0  00:00:00 bash

```

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

### 3-3-3 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:

$$\text{PRI}(\text{new}) = \text{PRI}(\text{old}) + \text{nice}$$

- 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。

### 3-3-4 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数据

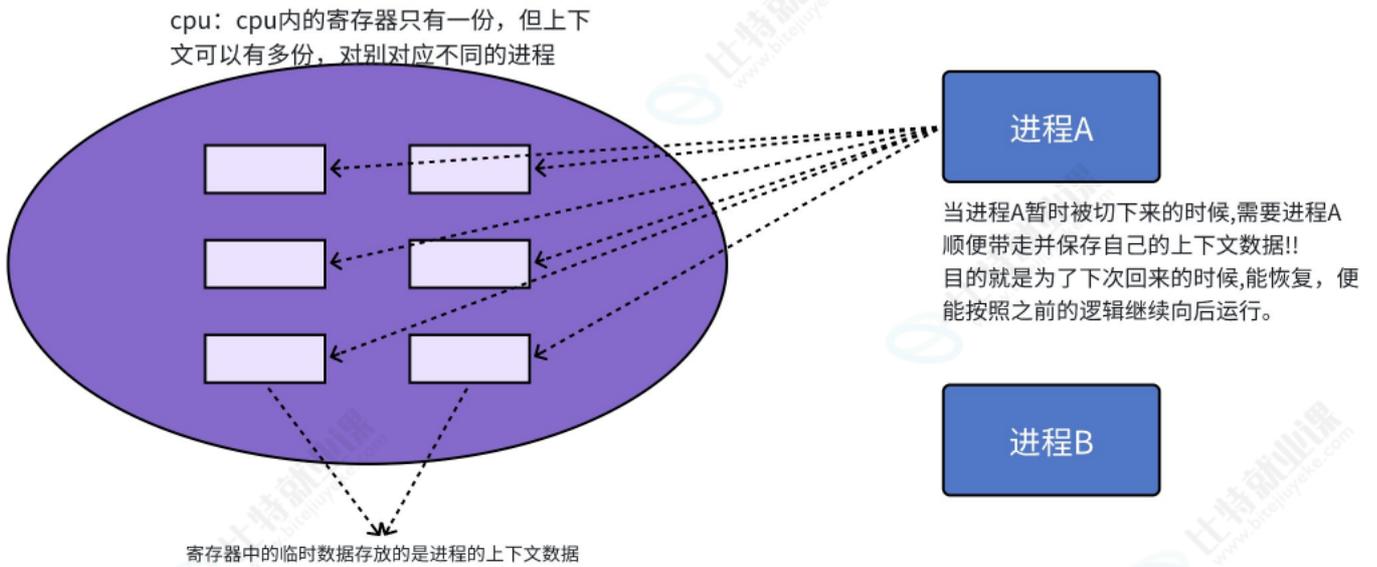
### 3-3-6 补充概念-竞争、独立、并行、并发

- 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

## 3.4 进程切换

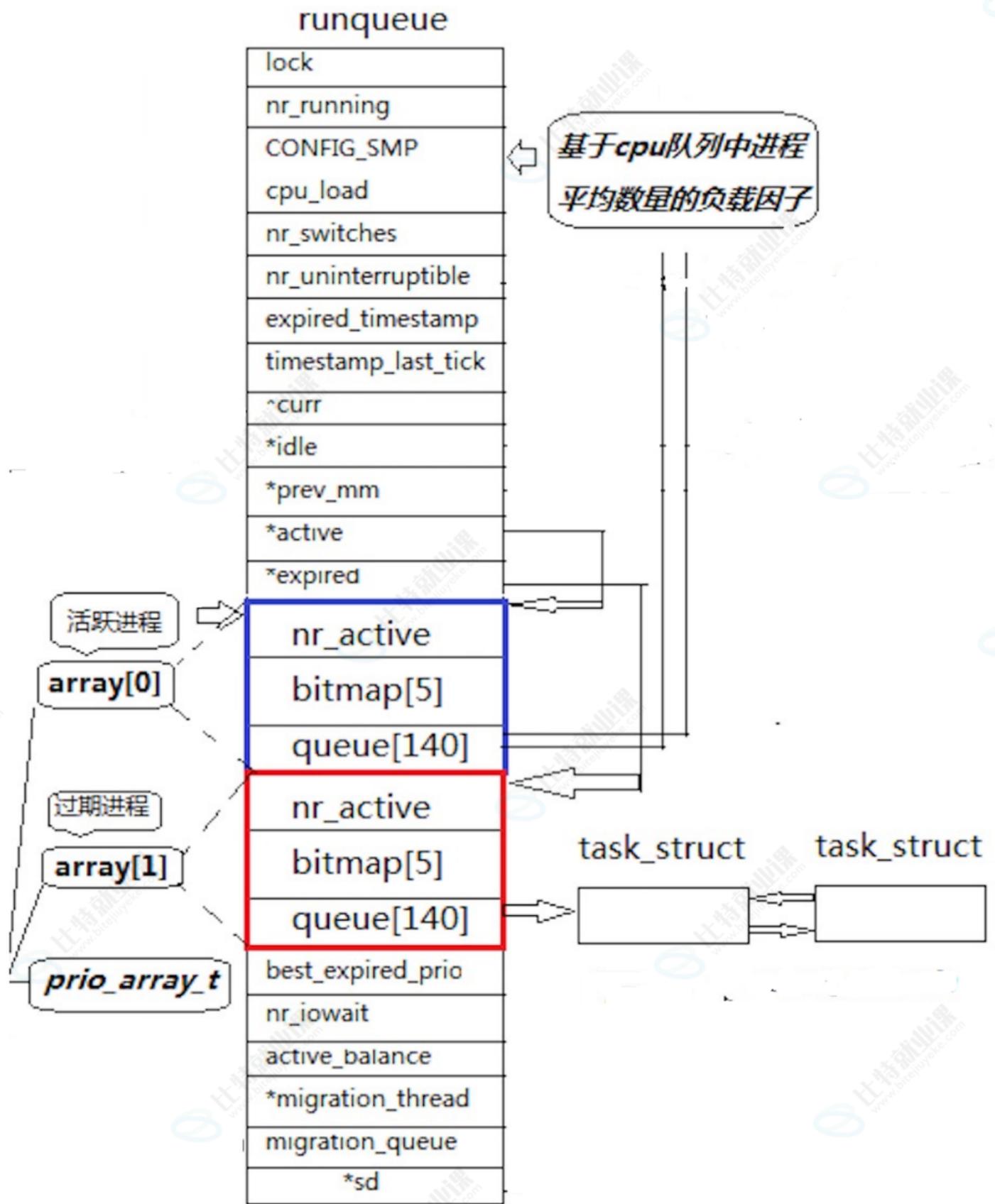
CPU上下文切换: 其实际含义是任务切换, 或者CPU寄存器切换。当多任务内核决定运行另外的任务时, 它保存正在运行任务的当前状态, 也就是CPU寄存器中的全部内容。这些内容被保存在任务自己的

堆栈中, 入栈工作完成后就把下一个将要运行的任务的当前状况从该任务的栈中重新装入CPU寄存器, 并开始下一个任务的运行, 这一过程就是context switch。



时间片: 当代计算机都是分时操作系统, 没有进程都有它合适的时间片(其实就是一个计数器)。时间片到达, 进程就被操作系统从CPU中剥离下来。

### 3-4 Linux2.6内核进程O(1)调度队列



### 3-4-1 一个CPU拥有一个runqueue

- 如果有多个CPU就要考虑进程个数的负载均衡问题

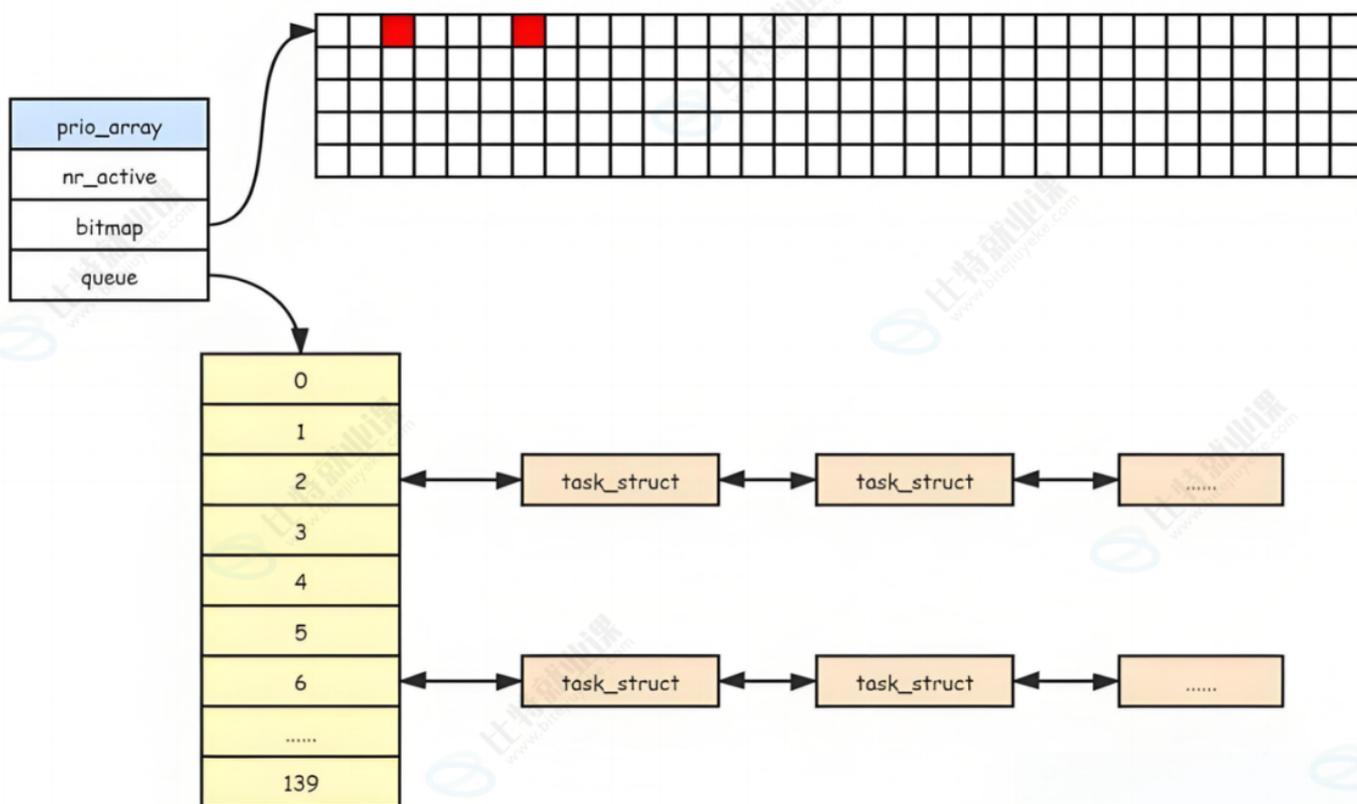
### 3-4-2 优先级

- 普通优先级：100~139（我们都是普通的优先级，想想nice值的取值范围，可与之应！）

- 实时优先级：0~99（不关心）

### 3-4-3 活动队列

- 时间片还没有结束的所有进程都按照优先级放在该队列
- nr\_active: 总共有多少个运行状态的进程
- queue[140]: 一个元素就是一个进程队列，相同优先级的进程按照FIFO规则进行排队调度,所以，数组下标就是优先级！
- 从该结构中，选择一个最合适的进程，过程是怎么的呢？
  1. 从0下标开始遍历queue[140]
  2. 找到第一个非空队列，该队列必定为优先级最高的队列
  3. 拿到选中队列的第一个进程，开始运行，调度完成！
  4. 遍历queue[140]时间复杂度是常数！但还是太低效了！
- bitmap[5]:一共140个优先级，一共140个进程队列，为了提高查找非空队列的效率，就可以用 5\*32 个比特位表示队列是否为空，这样，便可以大 提高查找效率！



### 3-4-4 过期队列

- 过期队列和活动队列结构一模一样
- 过期队列上放置的进程，都是时间片耗尽的进程
- 当活动队列上的进程都被处理完毕之后，对过期队列的进程进行时间片重新计算

### 3-4-5 active指针和expired指针

- active指针永远指向活动队列
- expired指针永远指向过期队列
- 可是活动队列上的进程会越来越少，过期队列上的进程会越来越多，因为进程时间片到期时一直都存在的。
- 没关系，在合适的时候，只要能够交换active指针和expired指针的内容，就相当于有具有了一批新的活动进程

## 4. 命令行参数和环境变量

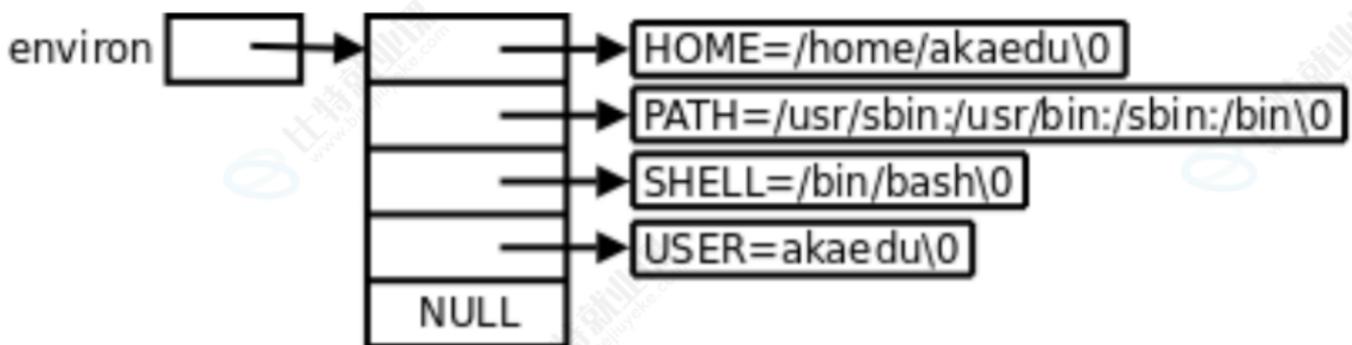
### 4-1 基本概念

- 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

### 4-2 常见环境变量

- PATH : 指定命令的搜索路径
- 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)
- SHELL : 当前Shell,它的值通常是/bin/bash。

### 4-5 环境变量的组织方式

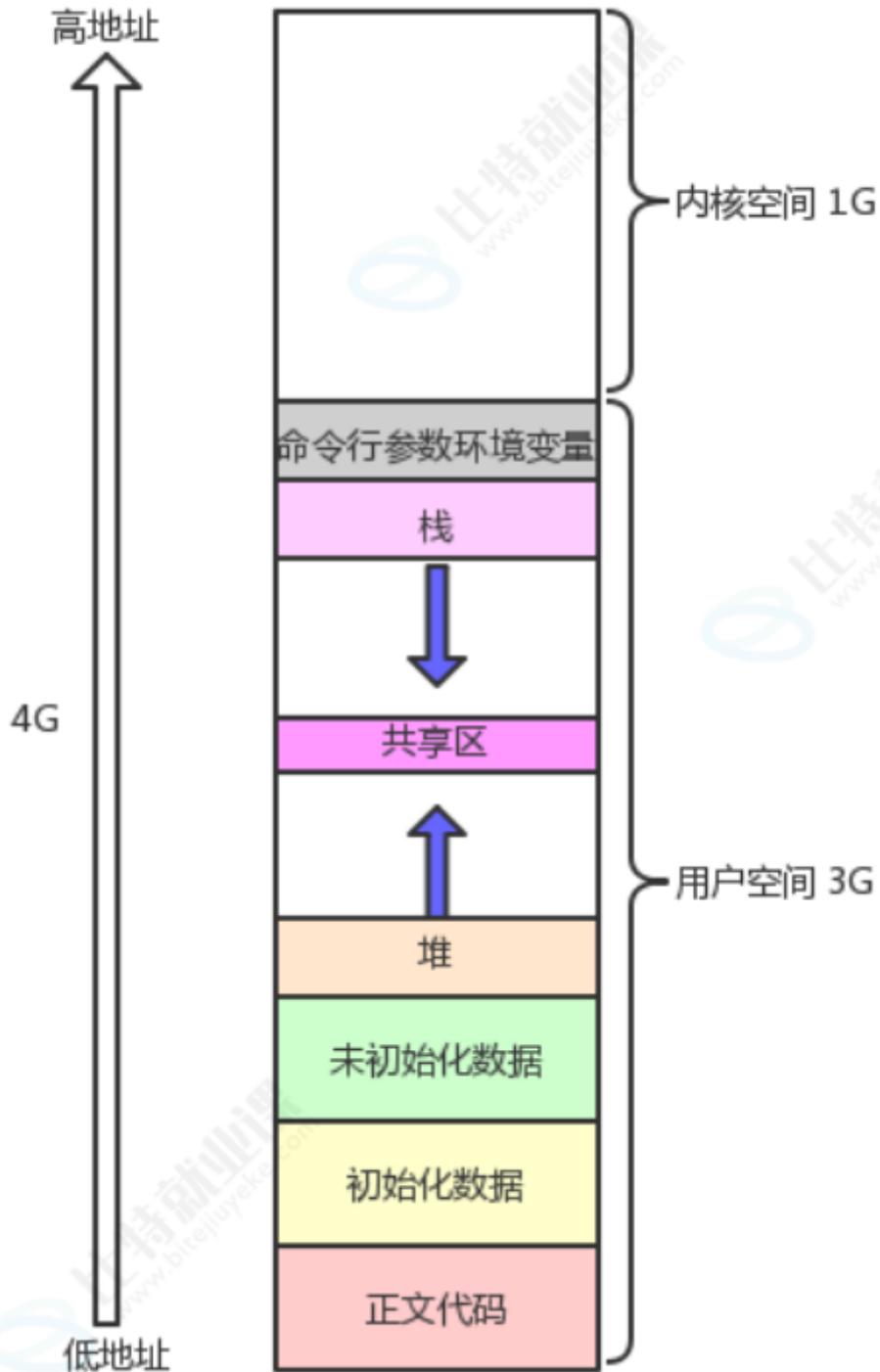


## 5. 程序地址空间

### 5-1 研究平台

## 5-2 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



代码块

```
1 #include <stdio.h>
2 #include <unistd.h>
```

```

3  #include <stdlib.h>
4  int g_unval;
5  int g_val = 100;
6
7  int main(int argc, char *argv[], char *env[])
8  {
9      const char *str = "helloworld";
10     printf("code addr: %p\n", main);
11     printf("init global addr: %p\n", &g_val);
12     printf("uninit global addr: %p\n", &g_unval);
13     static int test = 10;
14     char *heap_mem = (char *)malloc(10);
15     char *heap_mem1 = (char *)malloc(10);
16     char *heap_mem2 = (char *)malloc(10);
17     char *heap_mem3 = (char *)malloc(10);
18
19     printf("heap addr: %p\n", heap_mem);           // heap_mem(0), &heap_mem(1)
20     printf("heap addr: %p\n", heap_mem1);        // heap_mem(0), &heap_mem(1)
21     printf("heap addr: %p\n", heap_mem2);        // heap_mem(0), &heap_mem(1)
22     printf("heap addr: %p\n", heap_mem3);        // heap_mem(0), &heap_mem(1)
23     printf("test static addr: %p\n", &test);     // heap_mem(0), &heap_mem(1)
24     printf("stack addr: %p\n", &heap_mem);       // heap_mem(0), &heap_mem(1)
25     printf("stack addr: %p\n", &heap_mem1);      // heap_mem(0), &heap_mem(1)
26     printf("stack addr: %p\n", &heap_mem2);      // heap_mem(0), &heap_mem(1)
27     printf("stack addr: %p\n", &heap_mem3);      // heap_mem(0), &heap_mem(1)
28     printf("read only string addr: %p\n", str);
29
30     for (int i = 0; i < argc; i++)
31     {
32         printf("argv[%d]: %p\n", i, argv[i]);
33     }
34
35     for (int i = 0; env[i]; i++)
36     {
37         printf("env[%d]: %p\n", i, env[i]);
38     }
39
40     return 0;
41 }

```

#### 代码块

```

1  $ ./a.out
2  code addr: 0x40055d
3  init global addr: 0x601034
4  uninit global addr: 0x601040

```

```
5 heap addr: 0x1791010
6 heap addr: 0x1791030
7 heap addr: 0x1791050
8 heap addr: 0x1791070
9 test static addr: 0x601038
10 stack addr: 0x7ffd0f9a4368
11 stack addr: 0x7ffd0f9a4360
12 stack addr: 0x7ffd0f9a4358
13 stack addr: 0x7ffd0f9a4350
14 read only string addr: 0x400800
15 argv[0]: 0x7ffd0f9a4811
16 env[0]: 0x7ffd0f9a4819
17 env[1]: 0x7ffd0f9a482e
18 env[2]: 0x7ffd0f9a4845
19 env[3]: 0x7ffd0f9a4850
20 env[4]: 0x7ffd0f9a4860
21 env[5]: 0x7ffd0f9a486e
22 env[6]: 0x7ffd0f9a4892
23 env[7]: 0x7ffd0f9a48a5
24 env[8]: 0x7ffd0f9a48ae
25 env[9]: 0x7ffd0f9a48f1
26 env[10]: 0x7ffd0f9a4e8d
27 env[11]: 0x7ffd0f9a4ea6
28 env[12]: 0x7ffd0f9a4f00
29 env[13]: 0x7ffd0f9a4f13
30 env[14]: 0x7ffd0f9a4f24
31 env[15]: 0x7ffd0f9a4f3b
32 env[16]: 0x7ffd0f9a4f43
33 env[17]: 0x7ffd0f9a4f52
34 env[18]: 0x7ffd0f9a4f5e
35 env[19]: 0x7ffd0f9a4f93
36 env[20]: 0x7ffd0f9a4fb6
37 env[21]: 0x7ffd0f9a4fd5
38 env[22]: 0x7ffd0f9a4fdf
```

## 5-3 虚拟地址

代码块

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 int g_val = 0;
5
6 int main()
7 {
```

```

8     pid_t id = fork();
9     if (id < 0)
10    {
11        perror("fork");
12        return 0;
13    }
14
15    else if (id == 0)
16    { // child
17        printf("child[%d]: %d : %p\n", getpid(), g_val, &g_val);
18    }
19
20    else
21    { // parent
22        printf("parent[%d]: %d : %p\n", getpid(), g_val, &g_val);
23    }
24
25    sleep(1);
26
27    return 0;
28 }

```

#### 输出

```

1 //与环境相关, 观察现象即可
2 parent[2995]: 0 : 0x80497d8
3 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任何修改。可是将代码稍加改动:

#### 代码块

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4 int g_val = 0;
5
6 int main()
7 {
8     pid_t id = fork();
9     if (id < 0)
10    {
11        perror("fork");
12        return 0;

```

```

13     }
14
15     else if (id == 0)
16     { // 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
17         g_val = 100;
18         printf("child[%d]: %d : %p\n", getpid(), g_val, &g_val);
19     }
20
21     else
22     { // parent
23         sleep(3);
24         printf("parent[%d]: %d : %p\n", getpid(), g_val, &g_val);
25     }
26
27     sleep(1);
28
29     return 0;
30 }

```

输出

```

1 //与环境相关,观察现象即可
2 child[3046]: 100 : 0x80497e8
3 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:

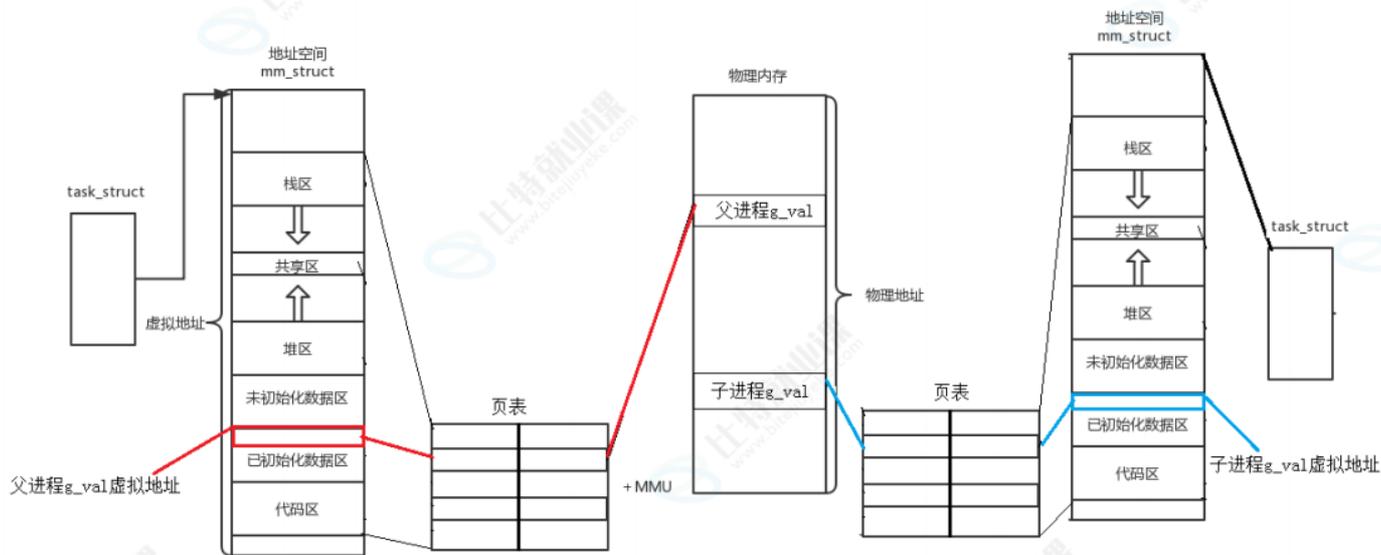
- 变量内容不一样,所以父子进程输出的变量绝对不是同一个变量
- 但地址值是一样的,说明,该地址绝对不是物理地址!
- 在Linux地址下,这种地址叫做 **虚拟地址**
- 我们在用C/C++语言所看到的地址,全部都是虚拟地址!物理地址,用户一概看不到,由OS统一管理

OS必须负责将 虚拟地址 转化成 物理地址。

## 5-4 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



好的，这两个概念是理解现代操作系统的基石。我们用一个生动的比喻来彻底讲清楚。

## 理解虚拟地址空间

### 核心比喻：每个进程都有自己的“私人宇宙”

想象一下，每个进程一出生，操作系统就为它分配了一个**完全独立的、完整的、从零开始寻址的线性内存宇宙**。

这个宇宙非常巨大（在 32 位系统上是 4GB，64 位系统上更是天文数字），并且**与所有其他进程的宇宙平行存在**。

代码块	
1 进程 A 的宇宙 (虚拟地址空间)	进程 B 的宇宙 (虚拟地址空间)
2 0x00000000	0x00000000
3 ↓	↓
4 ... [A的代码]	... [B的代码]
5 ... [A的数据]	... [B的数据]
6 ... [A的堆]	... [B的堆]
7 ↓	↓
8 0xFFFFFFFF	0xFFFFFFFF

**关键点：**

- 1. 独立性/安全性：**进程 A 无法直接感知或访问进程 B 的宇宙。一个进程的崩溃（宇宙爆炸）不会影响其他宇宙。这提供了根本的内存安全和稳定性。
- 2. 一致性/简化：**对于进程 A 而言，它“认为”自己独占了整个内存空间。它的代码、数据和堆栈总是出现在 **固定的、预期的地址** 上，这极大地简化了程序的编写和编译。
- 3. 欺骗性：**这个宇宙是“虚拟”的，它并不完全对应真实的物理内存（RAM）。操作系统和 CPU 内存管理单元（MMU）就像一个 **神奇的魔法部**，负责将虚拟宇宙中的地址翻译成真实的物理地址。这个翻译过程对进程是透明的。

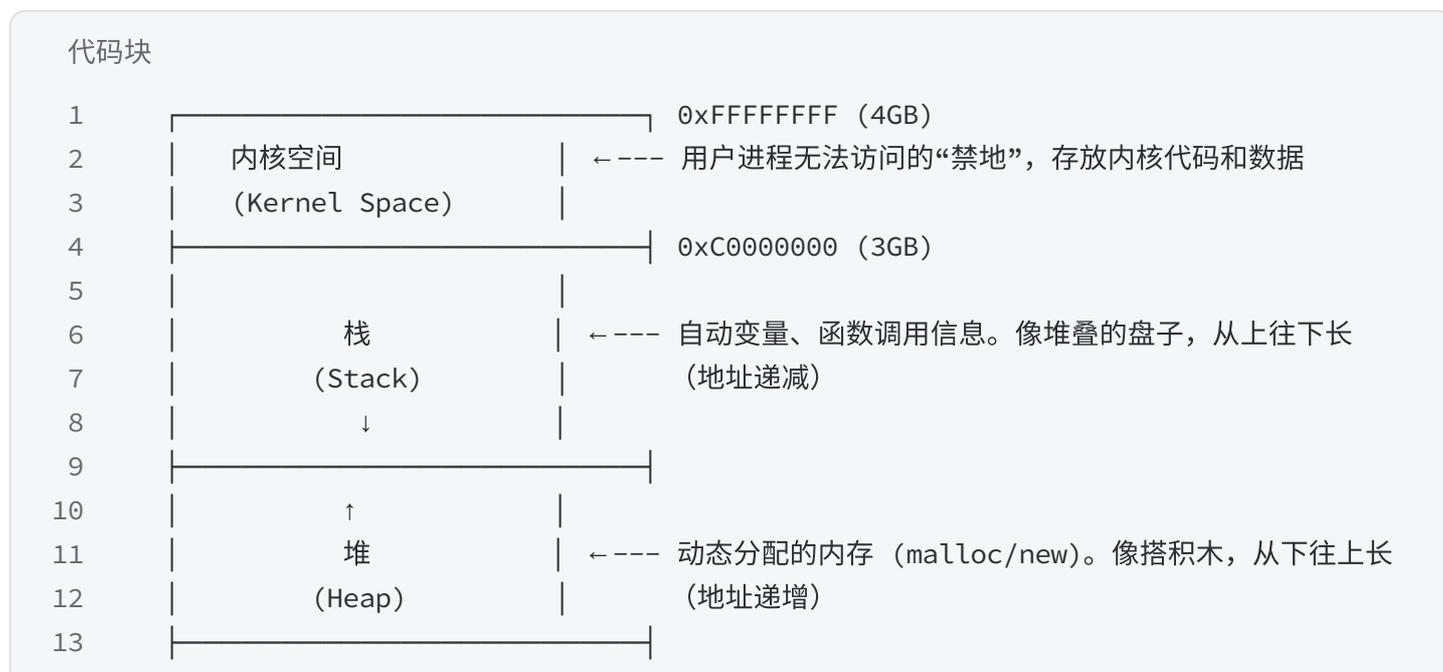
## 理解区域划分

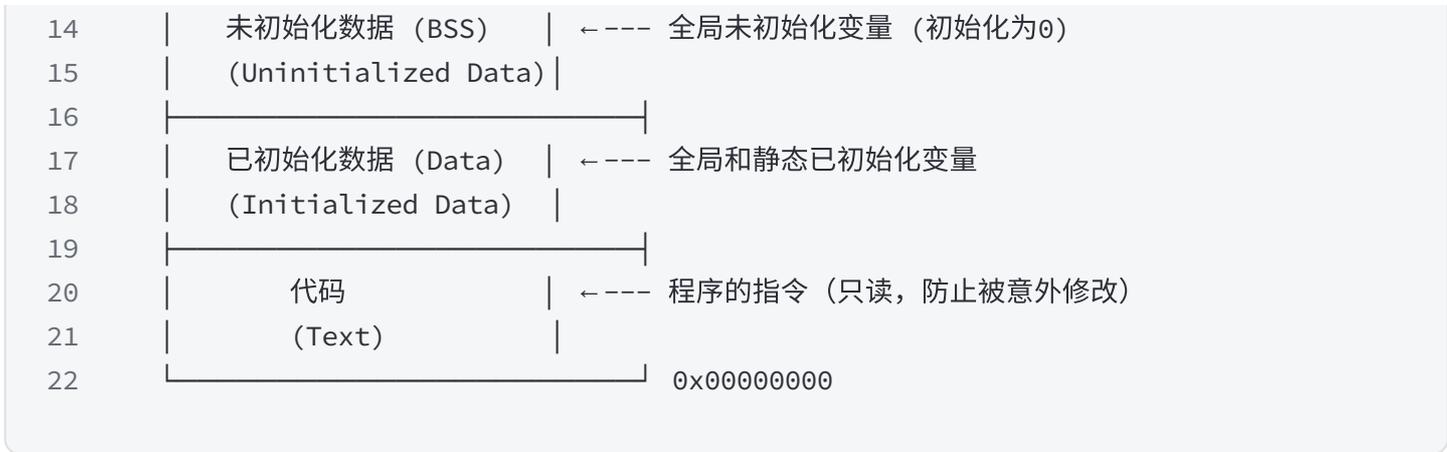
现在，我们来看看这个“私人宇宙”内部是如何规划和管理的。这就是 **区域划分**。

### 核心比喻：城市规划图

把虚拟地址空间想象成一块要开发的土地，操作系统是总规划师，它制定了严格的 zoning（功能分区）法律。

### 32 位 Linux 进程的经典城市规划：





## 为什么要这样划分？

### 1. 安全性 & 稳定性：

- **代码区只读**：防止程序指令被意外或恶意修改。
- **内核空间受保护**：用户程序崩溃不会波及操作系统。

### 2. 功能隔离 & 效率：

- **栈**：为函数调用设计，后进先出（LIFO），分配/释放极其高效（只是移动栈指针）。
- **堆**：为生命周期灵活的内存设计，可以按需分配和释放，但管理更复杂。
- **数据区**：让全局变量有固定的“家”，便于在程序启动时集中初始化。

### 3. 硬件协作：

- CPU 的 MMU 不仅做虚拟到物理的地址翻译，还负责强制执行这些区域的 **权限**（读、写、执行）。一旦程序越界访问（例如向代码区写入），MMU 会立即触发一个异常（段错误），由操作系统终止该进程。

## 总结

- **虚拟地址空间**：是一个 **美丽的谎言**。它给每个进程一个“独占整个内存”的假象，实现了安全、稳定和简化。
- **区域划分**：是这个谎言内部的 **精密结构**。它像城市规划一样，将内存按功能、生命周期和权限进行划分，确保了程序的高效、有序运行。

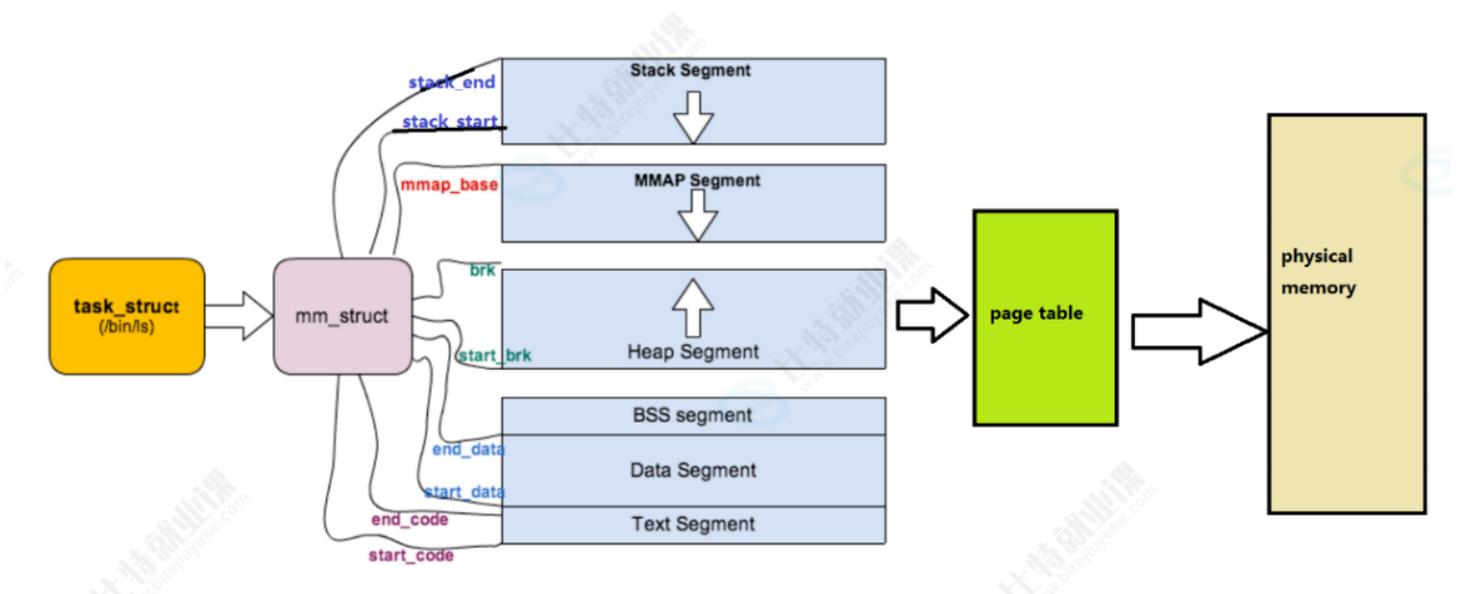
## 5-5 虚拟内存管理 - 第一讲

描述linux下进程的地址空间的所有的信息结构体是 `mm_struct`（内存描述符）。每个进程只有一个 `mm_struct` 结构，在每个进程的 `task_struct` 结构中，有一个指向该进程的 `mm_struct` 结构体指针。

代码块

```
1  struct task_struct
2  {
3      /*
4       * 对于普通的用户进程来说该字段指向他的虚拟地址空间的用户空间部分，
5       * 对于内核线程来说这部分为NULL。
6       */
7
8      struct mm_struct *mm;
9
10     /*
11     * 该字段是内核线程使用的。当该进程是内核线程时，它的mm字段为NULL，
12     * 表示没有内存地址空间，可也并不是真正的没有，
13     * 这是因为所有进程关于内核的映射都是一样的，内核线程可以使用任意进程的地址空间。
14     */
15
16     struct mm_struct *active_mm;
17 }
```

可以说，`mm_struct` 结构是对整个用户空间的描述。每一个进程都会有自己独立的 `mm_struct`，这样每一个进程都会有自己独立的地址空间才能互不干扰。先来看看由 `task_struct` 到 `mm_struct`，进程的地址空间的分布情况：



定位 `mm_struct` 文件所在位置和 `task_struct` 所在路径是一样的，不过他们所在文件是不一样的，`mm_struct` 所在的文件是 `mm_types.h`

代码块

```
1  struct mm_struct
2  {
3      /*...*/
4      struct vm_area_struct *mmap; /* 指向虚拟区间(VMA)链表 */
5      struct rb_root mm_rb;      /* red_black树 */
6      unsigned long task_size;   /*具有该结构体的进程的虚拟地址空间的大小*/
7      /*...*/
8      // 代码段、数据段、堆栈段、参数段及环境段的起始和结束地址。
9      unsigned long start_code, end_code, start_data, end_data;
10     unsigned long start_brk, brk, start_stack;
11     unsigned long arg_start, arg_end, env_start, env_end;
12     /*...*/
13 }
```

那既然每一个进程都会有自己独立的 mm\_struct，操作系统肯定是要将这么多进程的 mm\_struct 组织起来的！虚拟空间的组织方式有两种：

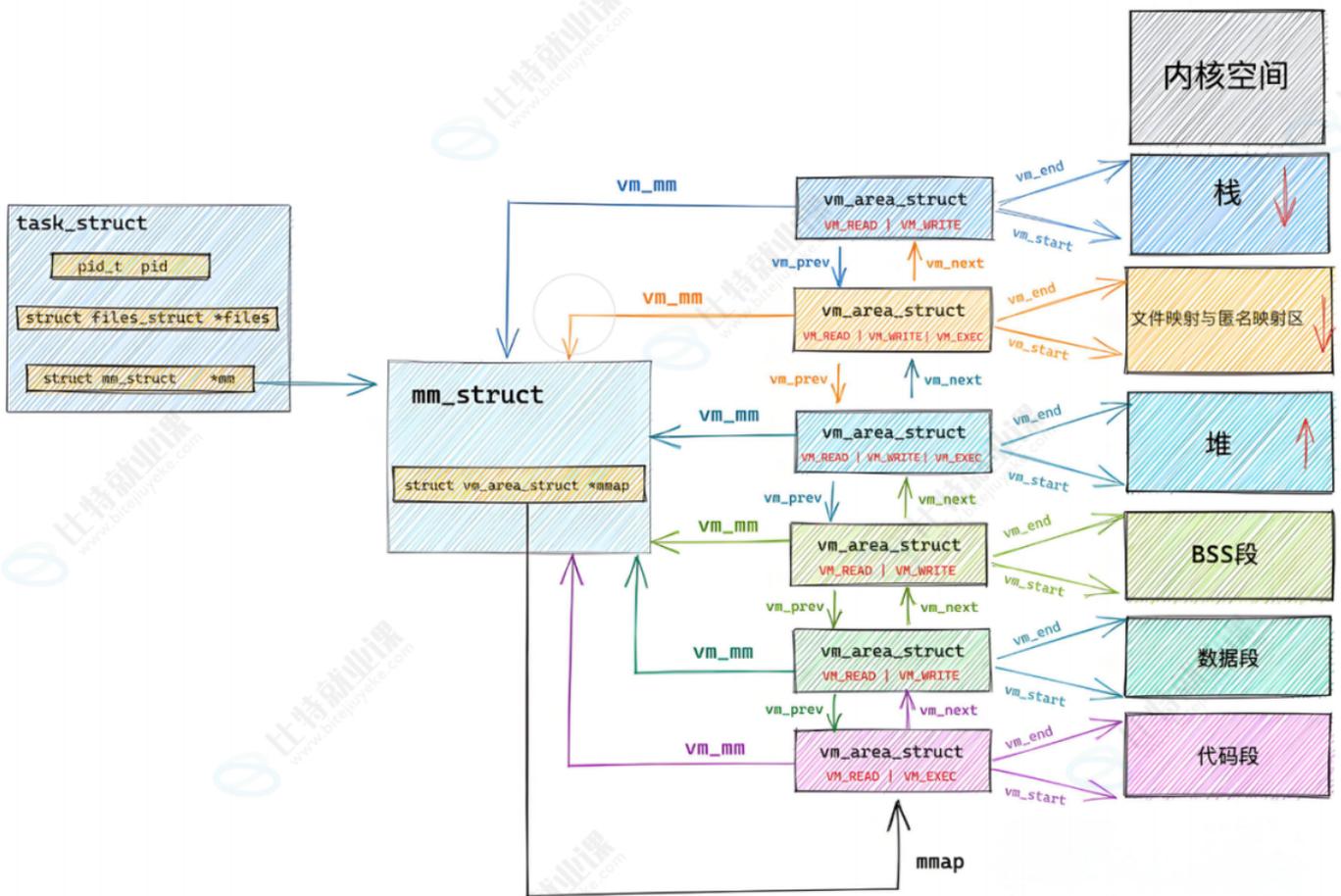
1. 当虚拟区较少时采取单链表，由mmap指针指向这个链表；
2. 当虚拟区间多时采取红黑树进行管理，由mm\_rb指向这棵树。

linux内核使用 `vm_area_struct` 结构来表示一个独立的虚拟内存区域(VMA)，由于每个不同质的虚拟内存区域功能和内部机制都不同，因此一个进程使用多个vm\_area\_struct结构来分别表示不同类型的虚拟内存区域。上面提到的两种组织方式使用的就是vm\_area\_struct结构来连接各个VMA，方便进程快速访问。

代码块

```
1  struct vm_area_struct
2  {
3      unsigned long vm_start;      // 虚存区起始
4      unsigned long vm_end;      // 虚存区结束
5      struct vm_area_struct *vm_next, *vm_prev; // 前后指针
6      struct rb_node vm_rb;      // 红黑树中的位置
7      unsigned long rb_subtree_gap;
8      struct mm_struct *vm_mm; // 所属的 mm_struct
9      pgprot_t vm_page_prot;
10     unsigned long vm_flags; // 标志位
11     struct
12     {
13         struct rb_node rb;
14         unsigned long rb_subtree_last;
15     } shared;
16     struct list_head anon_vma_chain;
17     struct anon_vma *anon_vma;
```



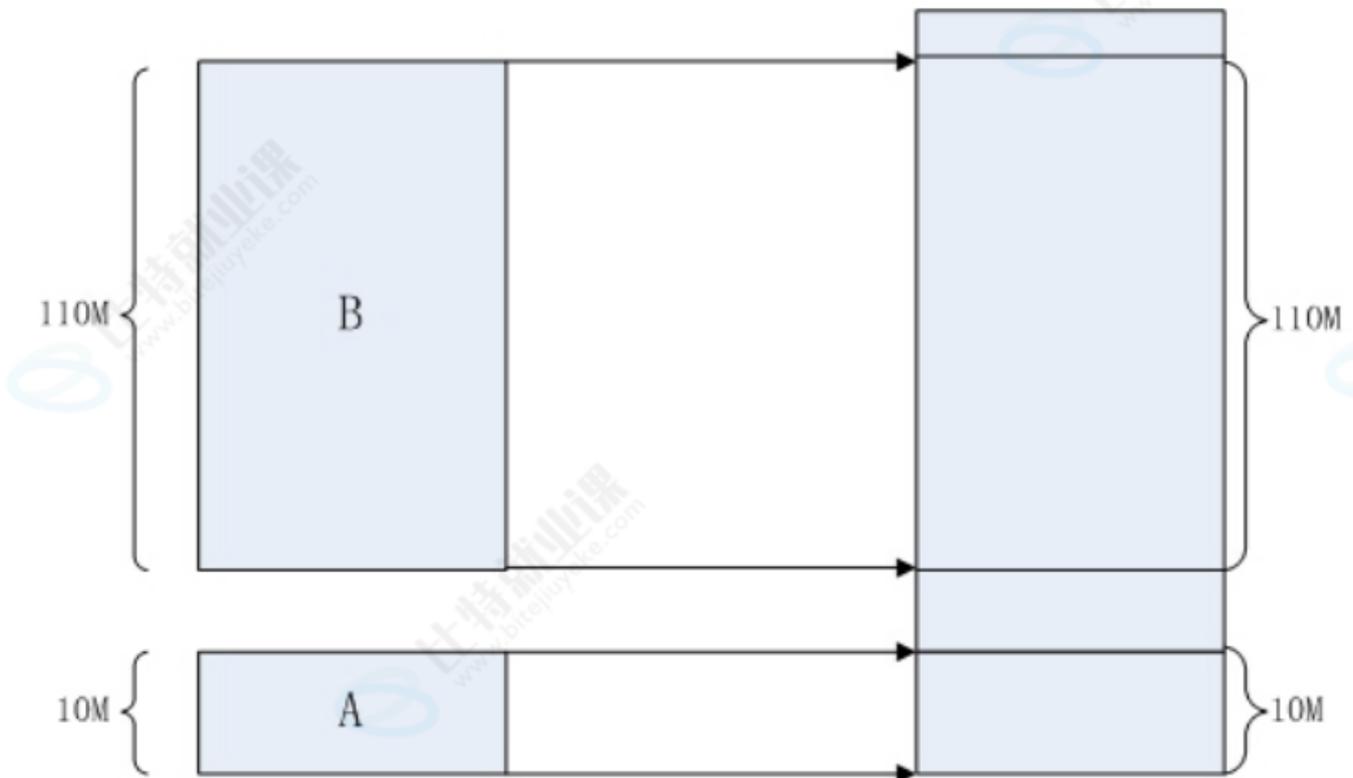


## 5-6 为什么要有虚拟地址空间

这个问题其实可以转化为：如果程序直接可以操作物理内存会造成什么问题？

在早期的计算机中，要运行一个程序，会把这些程序全都装入内存，程序都是直接运行在内存上的，也就是说程序中访问的内存地址都是实际的物理内存地址。当计算机同时运行多个程序时，必须保证这些程序用到的内存总量要小于计算机实际物理内存的大小。那当程序同时运行多个程序时，操作系统是如何为这些程序分配内存的呢？例如某台计算机总的内存大小是128M，现在同时运行两个程序A和B，A需占用内存10M，B需占用内存110。计算机在给程序分配内存时会采取这样的方法：先将内存中的前10M分配给程序A，接着再从内存中剩余的118M中划分出110M分配给程序B。

## 内存分配



这种分配方法可以保证程序A和程序B都能运行，但是这种简单的内存分配策略问题很多。

### · 安全风险

◦ 每个进程都可以访问任意的内存空间，这也就意味着任意一个进程都能够去读写系统相关内存区域，如果是一个木马病毒，那么他就能随意的修改内存空间，让设备直接瘫痪。

### · 地址不确定

◦ 众所周知，编译完成后的程序是存放在硬盘上的，当运行的时候，需要将程序搬到内存当中去运行，如果直接使用物理地址的话，我们无法确定内存现在使用到哪里了，也就是说拷贝的实际内存地址每一次运行都是不确定的，比如：第一次执行a.out时候，内存当中一个进程都没有运行，所以搬移到内存地址是0x00000000，但是第二次的时候，内存已经有10个进程在运行了，那执行a.out的时候，内存地址就不一定了

### · 效率低下

◦ 如果直接使用物理内存的话，一个进程就是作为一个整体（内存块）操作的，如果出现物理内存不够用的时候，我们一般的办法是将不常用的进程拷贝到磁盘的交换分区中，好腾出内存，但是如果是物理地址的话，就需要将整个进程一起拷走，这样，在内存和磁盘之间拷贝时间太长，效率较低。

物理内存是连续的,所以拷贝时需要一整块符合要求内存大小的空间,虚拟内存只用交换页就好了

了

存在这么多问题，有了虚拟地址空间和分页机制就能解决了吗？当然！

- 地址空间和页表是OS创建并维护的！是不是也就意味着，凡是想使用地址空间和页表进行映射，也一定要在OS的监管之下来进行访问！！也顺便 **保护了物理内存中的所有合法数据**，包括各个进程以及内核的相关有效数据！

- 因为有地址空间的存在和页表的映射的存在，我们的物理内存中可以对未来的数据进行任意位置的加载！物理内存的分配和进程的管理就可以做到没有关系，**进程管理模块和内存管理模块就完成了解耦合**。

- 因为有地址空间的存在，所以我们在C、C++语言上new, malloc空间的时候，其实是在地址空间上申请的，物理内存可以甚至一个字节都不给你。而当你真正进行对物理地址空间访问的时候，才执行内存的相关管理算法，帮你申请内存，构建页表映射关系（延迟分配），这是由操作系统自动完成，用户包括进程完全0感知！！

- 因为页表的映射的存在，程序在物理内存中理论上就可以任意位置加载。它可以将地址空间上的虚拟地址和物理地址进行映射，在 **进程视角所有的内存分布都可以是有序**的。

虚拟空间连续,但是物理空间不连续