阻塞等待本质，挂起等待的含义

创建一个进程先建立内核数据结构

（进程新建）建立内核数据结构----进程就存在了（只是不会被调度）因为没有代码和数据

（进程就绪）已经创建了内核数据结构并且也加载了代码和数据。

在linux 操作系统两种状态很瞬时，因此看成一个，操作系统释放都时候是先代码和数据再内核数据结构

我想看到僵尸状态如何设计代码，

创建子进程-》父子同时存在-》让子进程退出，父进程还存活，但是父进程什么都不做  
父子进程的关系？？  
defunct非现存的，失灵的，不再使用的；死的

设计一个僵尸进程

子进程一直处于僵尸状态，子进程task\_struct就不会回收，僵尸进程积累到一定程度就会造成内存泄露。

阻塞等待和挂起等待

阻塞等待

是进程或线程**主动让出CPU资源并进入睡眠状态**，直到特定条件满足后才被唤醒继续执行。其核心目的是**避免CPU空转浪费资源**​（即避免“忙等待”），实现高效的并发管理。

**核心机制：等待队列 (Wait Queue)​​**

1. ​**注册等待项**​  
   当进程/线程需要等待某个事件（如数据到达、资源就绪、锁释放等）时：
   * 将自身加入与事件关联的**内核等待队列**​（wait\_queue\_head\_t）。
   * 设置自身状态为 TASK\_INTERRUPTIBLE（可中断睡眠）或 TASK\_UNINTERRUPTIBLE（不可中断睡眠）。

**主动让出CPU**​  
调用 schedule() 触发**调度器**​

set\_current\_state(TASK\_INTERRUPTIBLE); // 设置睡眠状态

schedule(); // 主动放弃CPU，切换到其他进程

**唤醒过程：条件满足时​**

当事件发生（如I/O完成、锁释放）时：

1. ​**事件触发者**​（例如中断处理程序、释放锁的进程）调用 wake\_up() 系列函数：
2. ​**内核将等待队列中的进程状态改为**TASK\_RUNNING，并移出等待队列。
3. ​**调度器**在下次调度时，这些进程重新获得执行机会。

**关键特点与本质​**

1. ​**非忙等待 (Non-Busy Waiting)​**​
   * 阻塞时**完全不消耗CPU**，进程处于睡眠状态（Sleeping），资源仅保留在内存中。
   * 对比“忙等待”（如 while (!condition);）会持续占用CPU空转。
2. ​**依赖调度器协调**​
   * 进程唤醒后**不会立即执行**，需等待调度器分配CPU时间片（可能被更高优先级任务抢占）。
3. ​**状态可追踪**​
   * 在 ps 命令中：
     + ​\*\*S\*\*​：可中断睡眠（通常等待事件，如读Socket）。
     + ​\*\*D\*\*​：不可中断睡眠（通常等待磁盘I/O，不能被信号中断）。
4. ​**与中断/事件的紧密联动**​
   * 阻塞等待通常由**硬件中断**​（如网络数据到达、磁盘读取完成）触发唤醒。

这种机制是Linux实现**高并发、低资源消耗**的核心基础，也是理解同步I/O（如read/write）、锁（mutex/semaphore）等行为的关键

| **特性​** | **​阻塞​** | **​非阻塞​** |
| --- | --- | --- |
| 行为 | 等待条件满足才返回 | 立即返回（无论条件是否满足） |
| CPU使用 | 睡眠时不消耗CPU | 需轮询检查条件（可能忙等待） |
| 实现方式 | 依赖等待队列 + schedule() | 返回 EAGAIN/EWOULDBLOCK |
| 适用场景 | 同步I/O、锁操作 | 高并发异步模型（如epoll） |

挂起等待（Suspended Waiting）是指**进程/线程因资源不足或外部干预，被强制移出内存并暂停执行，但其等待状态仍被保留**，直到满足条件后重新激活。这是一种**深度休眠状态**，与阻塞等待的关键区别在于**是否占用内存资源**。

| **特性​** | **阻塞等待 (Blocking)** | **挂起等待 (Suspended)** |
| --- | --- | --- |
| ​**内存占用**​ | 保留在内存中 | 数据被移至磁盘 |
| ​**触发原因**​ | 进程主动等待事件 | 系统资源不足/外部强制干预 |
| ​**恢复成本**​ | 直接调度（微秒级） | 需磁盘I/O加载数据（毫秒级） |
| ​**状态可见性**​ | S（睡眠）或 D（不可中断） | T（停止）或 D+（不可中断+挂起） |
| ​**能否被信号中断**​ | 可中断状态（S）响应信号 | 通常不响应信号（除非手动恢复） |

**核心机制解析**

**1. ​挂起的触发原因​**

* ​**内存资源不足**​（最常见场景）：  
  当系统物理内存严重不足时，内核通过**交换机制 (Swapping)​**​ 将部分进程的数据（代码、堆栈等）从内存转移到磁盘交换区（Swap Space），释放内存供活跃进程使用。
* ​**手动挂起**​：  
  用户通过命令（如 kill -SIGSTOP <PID>）或调试器暂停进程执行。
* ​**任务调度策略**​：  
  如实时系统的低优先级任务被高优先级任务抢占时可能被挂起。

这种机制是Linux实现**高并发、低资源消耗**的核心基础，也是理解同步I/O（如read/write）、锁（mutex/semaphore）等行为的关键。

语言层面的内存泄露问题，如果在常驻内城区的进程中出现，影响比较大

父进程退出，子进程在-----子进程会被系统自动领养—在后台运行-----孤儿进程

1. initd/systemd----系统

Linux系统中t与T状态

一个进程被加载到系统里立刻就处于就绪状态，Linux系统中新建，就绪状态不做区分。

进程优先级是什么为什么，怎么办？

是什么---获得某种资源的先后顺序---比如排队的本质就是在确认优先级

资源---打饭的窗口

为啥

本质其实就是目标资源比较少

Cpu

外设

这些资源虽然看着多，但是对于进程来讲都是少量的

优先级vs权限

一般权限问题解决能还是不能的问题，优先级潜台词是已经能了，只不过是谁先谁后的问题。

怎么办？？

Task\_struct -> 优先级属性->特定的几个int类型的变量表示优先级------优先级数字越小，代表优先级越高

Ps -al 可以查看linux进程的优先级

Pri标题下是进程优先级

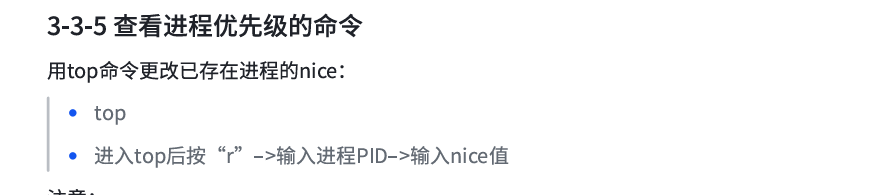
Nice(细微的)nice值：优先级的nice数据

最终优先级=pri(默认/老的/固定的基础优先级)+nice

为啥不直接改pri而是要整一个修正数据，因为如果一个进程正在调度，把优先级调高来，他算高还是算低，不好判断

因此我们用nice值来修改优先级，等操作系统再次调度该进程的时候，就会更新该进程的进程优先级，然后再对该进程进行调度

UID表示当前进程的用户id



字符串做修改还是比较时间复杂度都很高，因此在操作系统用户除了用户名·还会被分配一个用户id(数字).

操作系统对于用户是不是同一个用户也是对比的数字而不是名字

我们新建的任何文件，操作系统怎么知道文件/目录属于谁啊，他是通过给这个文件写入你的用户id 来标识的。

通过uid 操作系统不仅能识别出文件是谁创建的，而且还能识别出进程是谁启动的。

文件会记录下拥有者，所属组。和对应的权限

Linux一切皆文件

所有操作都是进程操作，进程自己会记录谁启动的我

因此当我们用进程去控制与文件相关的概念时就可以通过进程自己对应的uid来和文件的拥有者所属组id做对比，就可以进行权限控制，权限控制实现的原则，文件可追溯，进程可追溯。

进程记录下来了我，知道我是谁，文件也记录下了我和其他人知道我和其他人是谁，你进程要访问文件，操作系统会检测uid是不是相等，在不在所属用户和用户组，

有无读写权限，无读没写直接删掉

进程优先级：是为了竞争cpu资源。

怎么办

如何修改优先级

1指令

2代码

3修改进程优先级不是一个特别高频的动作，而且不建议修改

Top ->r->按照提示修改即可（os禁止频繁修改，没有权限修改）nice>=-20,nice<=19

nice>=-20,nice<=19

Nice可取40个数字

为啥nice值在可控范围内？

分时系统—进程调度时要尽量公平，因为在可控范围内才不会影响我系统的公平调度，如果不加控制的话，当用户把一个进程的优先级调的特别高就会产生进程饥饿现象

进程切换

1. 概念储备

A 在现代操作系统里每一个进程都有自己的时间片，时间片到了，进程就要被切换，无时间片，一个进程跑死循环他就能一直占用cpu

B linux是基于时间片调度轮转的，

C 一个进程在时间片到来的时候，并不一定跑完了，可以在任何一个地方被重新调度切换。

进程如何切换？？

进程如何切换？？

感性理解进程切换

在校入伍

给学校打招呼---保留学籍

在部队历练

回到学校---给学校打招呼—恢复学籍

**保存上下文是操作系统的职责，进程自己完全不知情也无法操作**。

进程就是我 学校是OS

选去当兵就是时间片到了

给学校打招呼---保留学籍—保留上下文

回到学校---给学校打招呼—恢复学籍---恢复上下文

7.15.2025

2切换进程和理解

* 1. 你的进程在运行的时候，会有很多的临时数据，都在cpu 的寄存器中保存

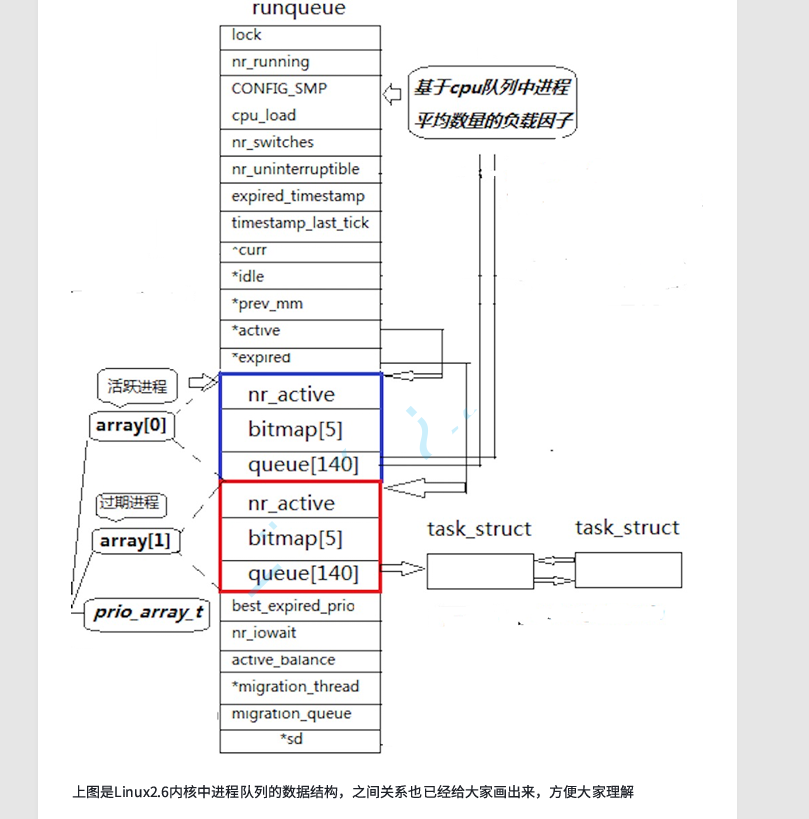
Pc：当前正在执行指令的下一条指令的地址

Ir:指令寄存器，就是正在执行的指令

* 1. **cpu内部的寄存器的数据是进程执行时的瞬时状态信息数据---上下文数据**

**正在系统中执行的进程被切走，系统中的cpu 会保存他的临时数据，cpu会将我们所关心的用户级的数据保存起来，如果不保存，下一个在进程的临时数据就会把上一个进程在寄存器中的临时数据搞乱**

**进程切换核心就是上下文数据的保存和切换**

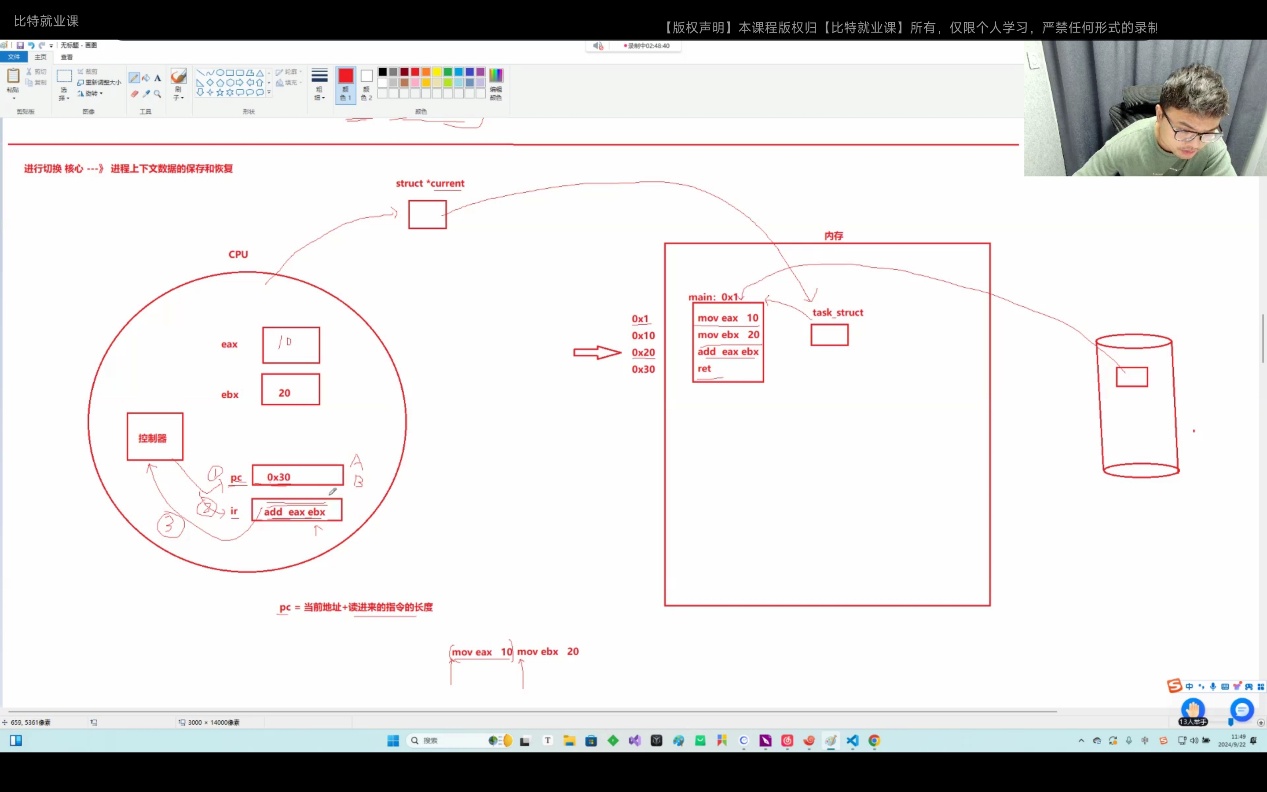
****

**运行队列**

**Active**

**Expired**

**7/16/2025**

****

**首先将高级编程语言转为汇编语言，一条高级语言会被转为很多条汇编语言。首先将第一条语句的内存地址传入pc寄存器，ir寄存器根据pc寄存器中的地址将第一条指令加载进ir寄存器，与此同时pc寄存器中的地址加载为addr(pc)=当前地址+读进来的指令的地址长度。接着，ir寄存器中的指令交给控制器执行。然后ir寄存器再次根据pc寄存器中的地址，将第二条指令加载进来，循环往复**

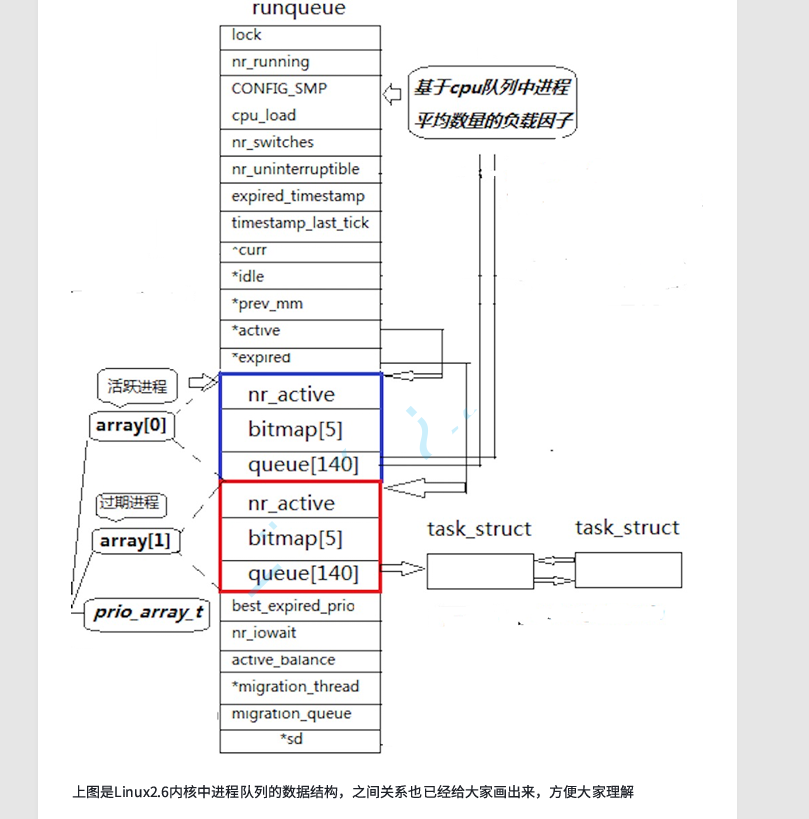
**保存上下文的地方在进程自己的pcb中的tss中**

**调度**

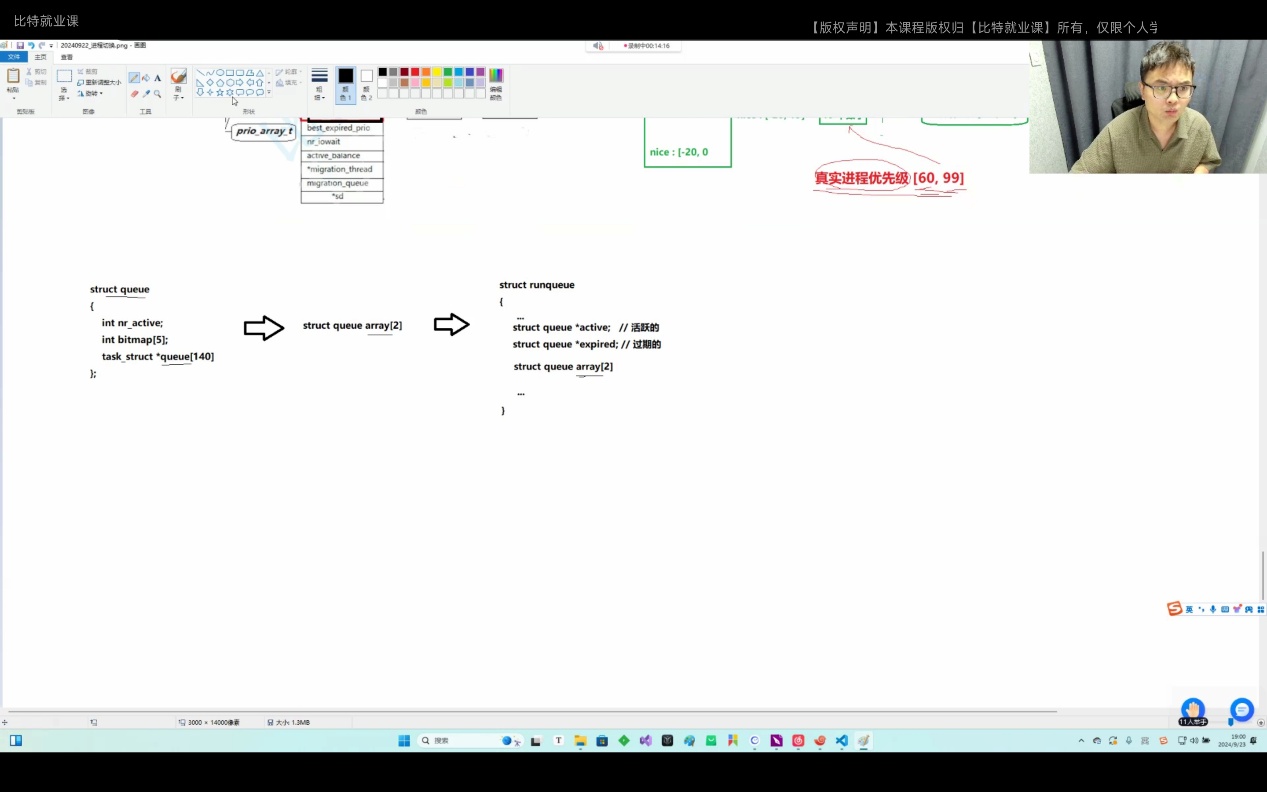
**7.16.2025**

**fork系统调用通过复制父进程创建一个子进程，父子进程数据独有，代码共享（在数据不发生改变的情况下父子进程资源指向同一块物理内存空间（调研写时拷贝技术））**

****

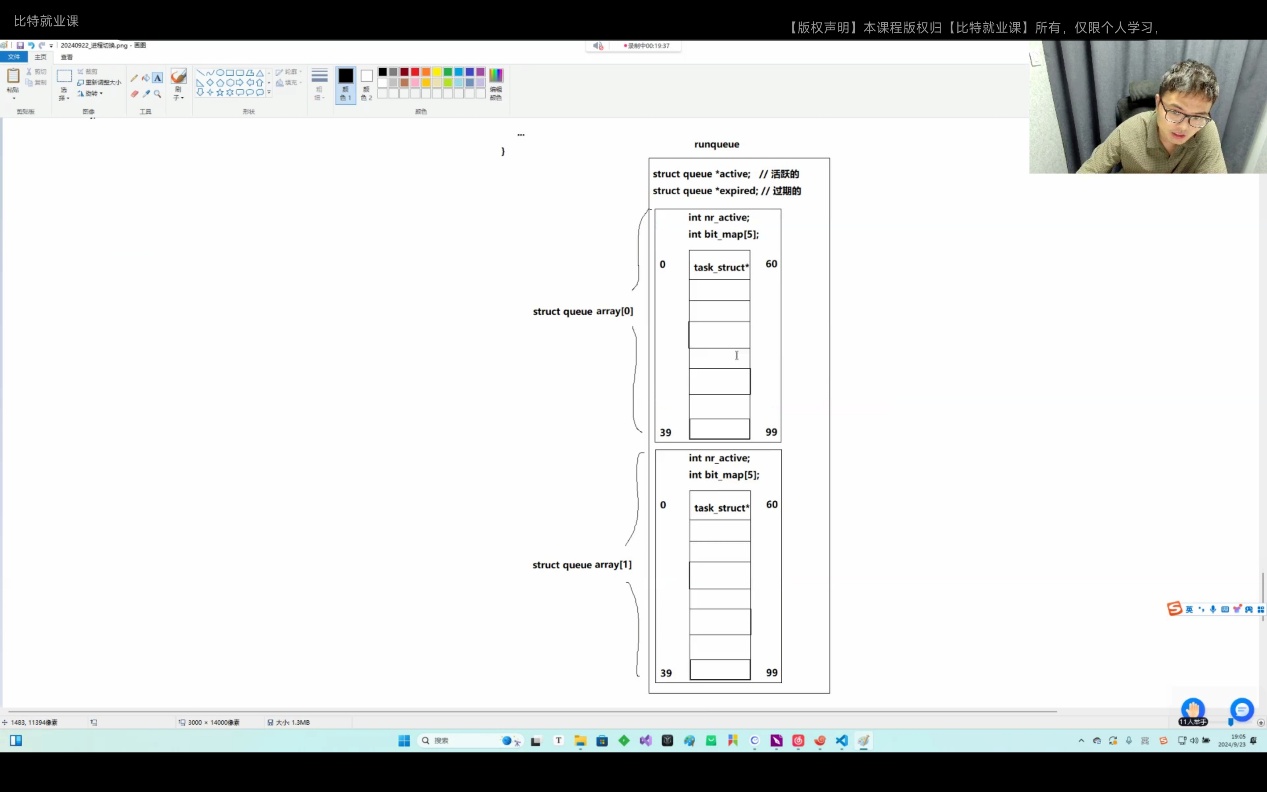
****

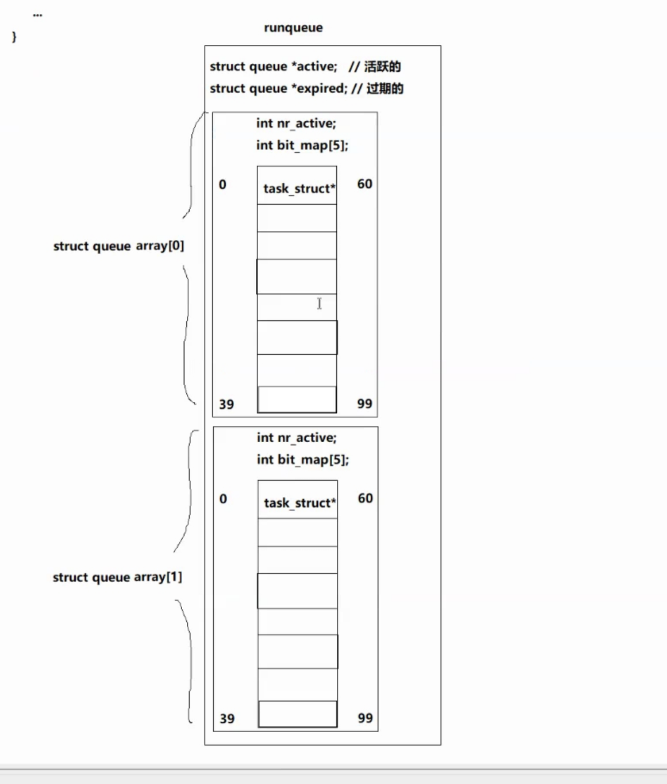
**每一个cpu都要有自己的运行队列，在cpu自己的队列中hash表有两张**

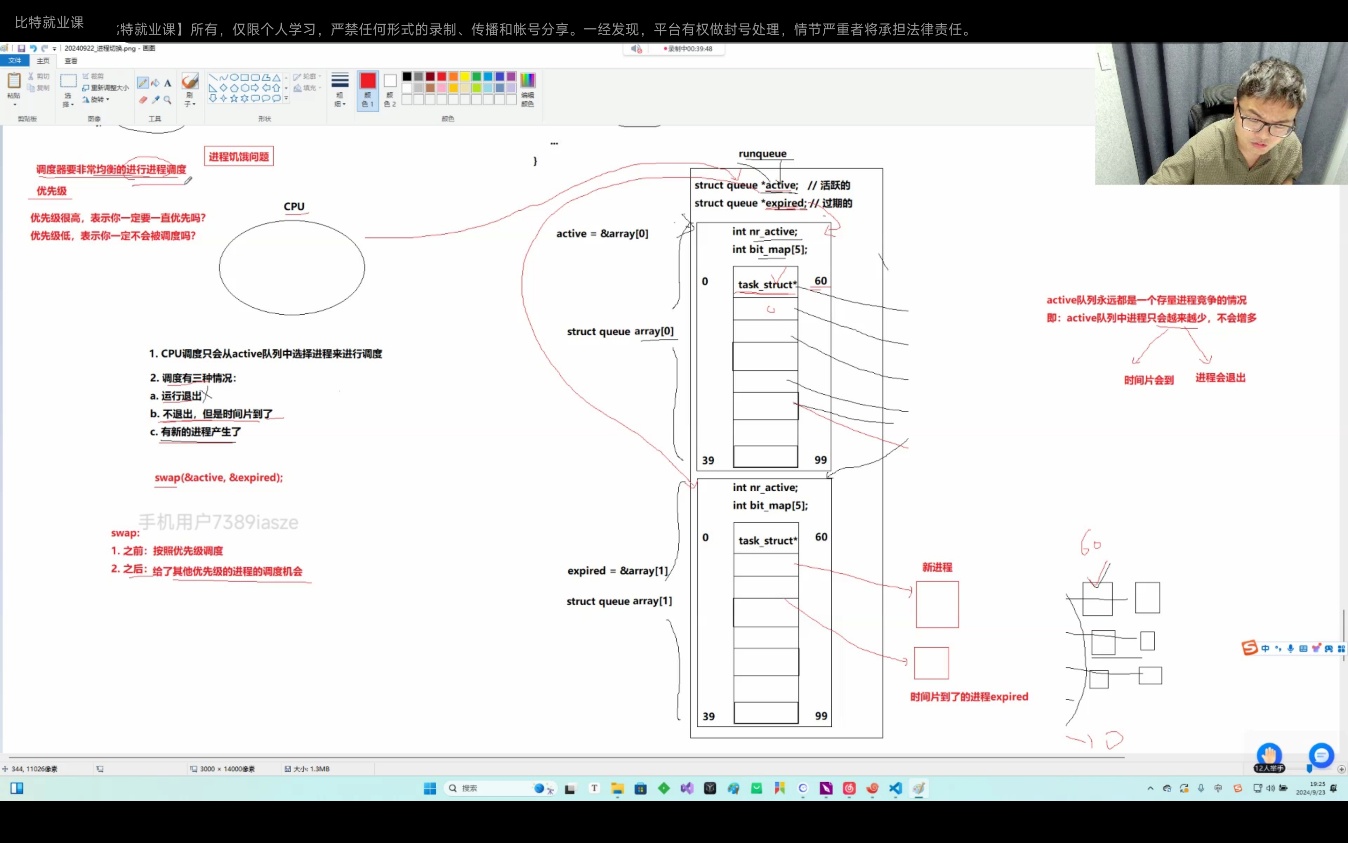
****

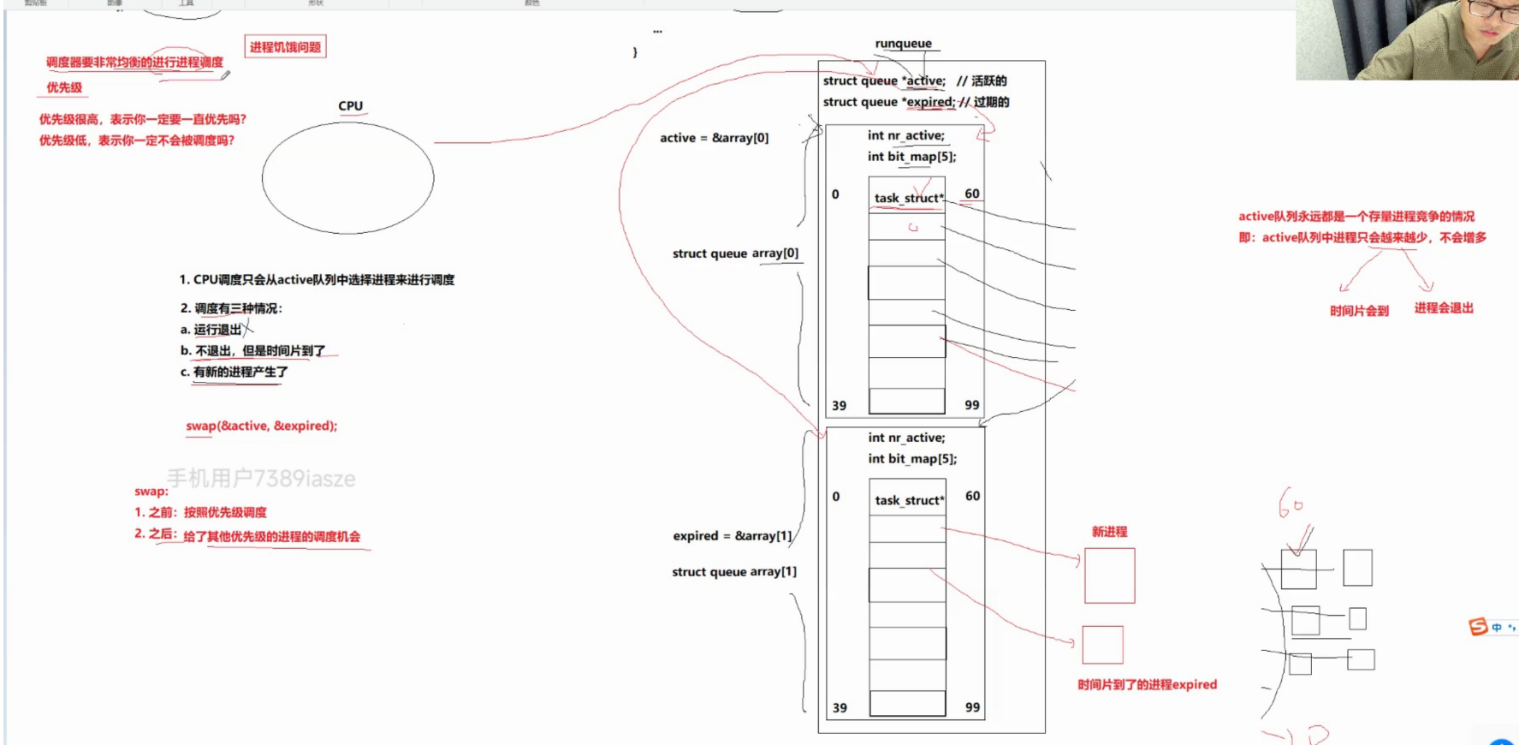
**Nr\_active表示队列中有多少个进程，就是hash表里有多少个进程，决定了要不要交换**

**Bitmap[5]表示？？？**

****

****

****

****

Struct queue\*active指向array[0], Struct queue\*expired指向array[1]

需要已知的前提

1，CPU调度只会从active队列中选择进程来进行调度

2，调度有三种情况

A. 运行退出---不会再进行调度，不考虑

B. 不退出，但是时间片到了

C. 有新的进程产生了

3， 调度器要非常均衡的进行进程调度这似乎与我们所熟知的优先级冲突了

优先级是优先级高的进程先调度，优先级低的后调度。

我们如果只维护一个队列的话，肯定会产生进程饥饿现象，因为较高优先级的进程时间片到了但是不退出，他就仍会排在active queue的前面，或者是有新的进程产生且进程的优先级较高。这样就会导致队列后面优先级低的一直不能被调度，这样就会导致进程饥饿，除此之外，当我们进程进程调度的时候，此时如果有新的进程产生，新的进程让操作系统通过runing queue对该进程进行排队，而cpu让操作系统对runing queue中的进程进行调度。这样会大大降低进程调度的效率

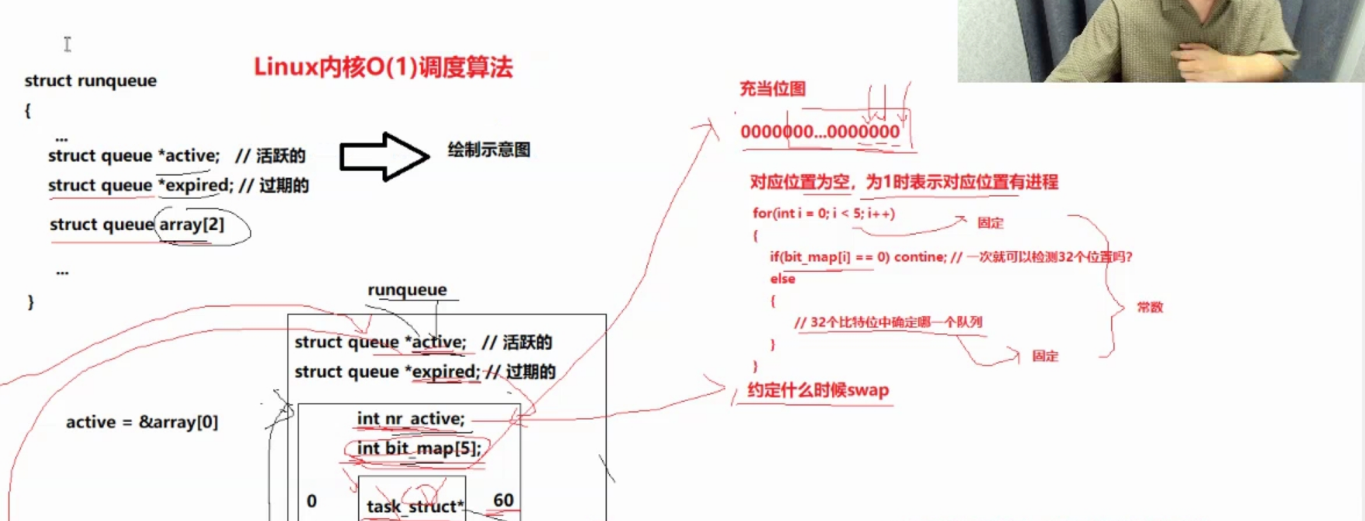
怎么避免进程饥饿呢？？

我们的runing queue并不是只有一个队列，而是两个。当产生新的进程或者时间片耗尽的进程但不退出时，我们将进程放入expired指向的Struct queue。

这样的话active队列中的进程数会越来越少，不会增多，操作系统会根据Nr\_active来交换active和expired如此一来，就会让保证根据进程优先级来进行调度的同时，保证较低优先级的进程也会被调度。

Active要进行调度的时候首先要判断nr\_active是不是等于0，如果nr\_active等于0直接无脑swap.cpu 调度的时候要

**Bitmap【5】充当位图**

****

**在通过**

**Bit\_map为整形数组5\*32=160。正好覆盖queue[140]。**cpu让操作系统对runing queue中的进程进行调度时,需要遍历**queue[140]，时间复杂度为O(n);**

**但是还是不太行，加一个bitmap, Bit\_map为整形数组5\*32=160。正好覆盖queue[140]。通过上述图中都for循环，最坏情况下时间复杂度为32\*5=160；**

**补充知识：**

**Struct node**

**{**

**Struct node\*next;**

**Struct node\*prev;**

**}**

**Struct node**

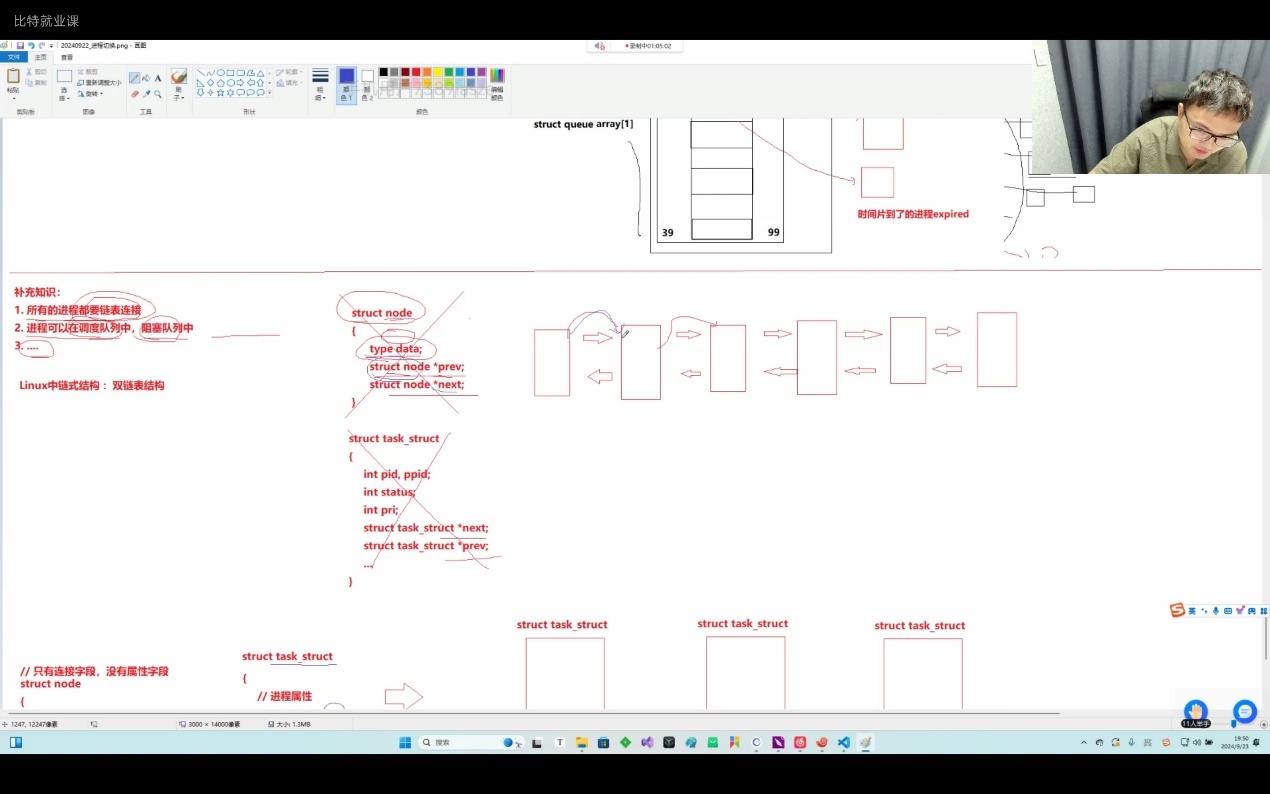
**{**

**//进程属性**

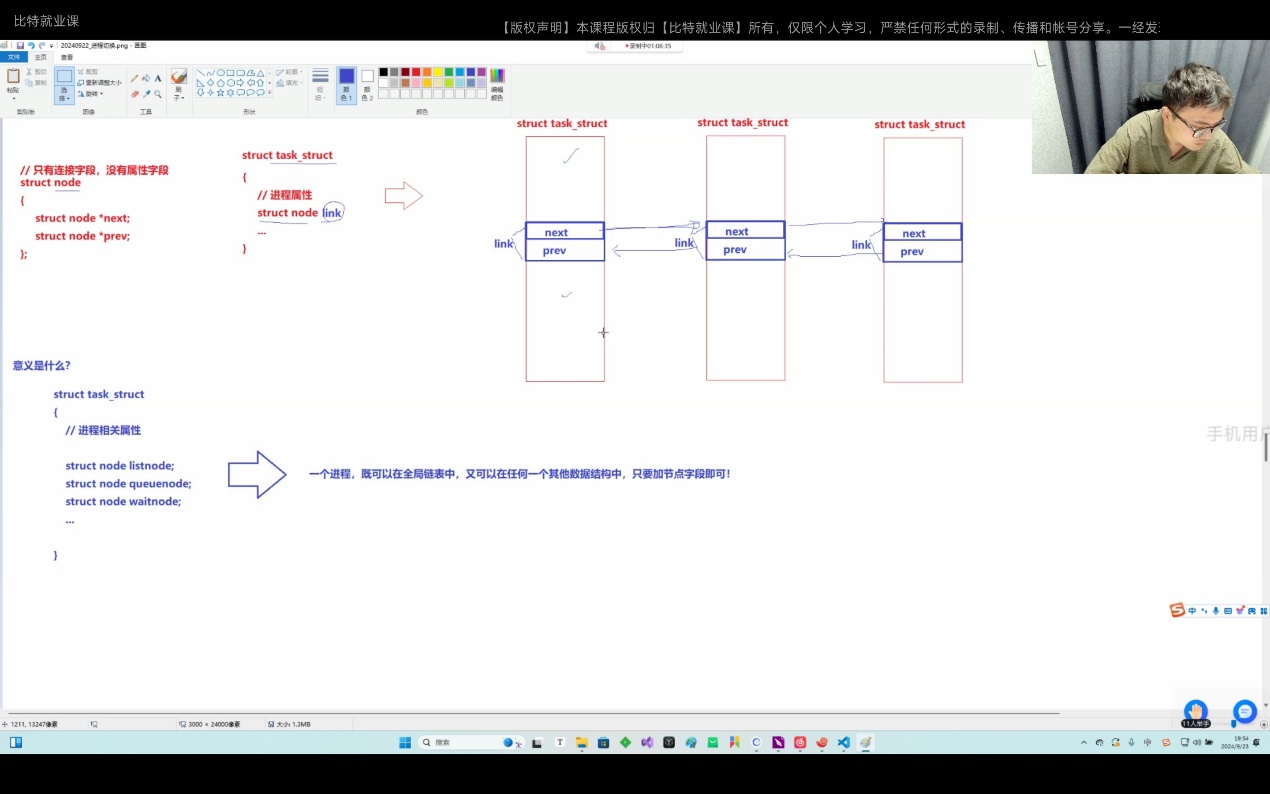
**……**

**Struct node;**

**}**

****

**图一**

****

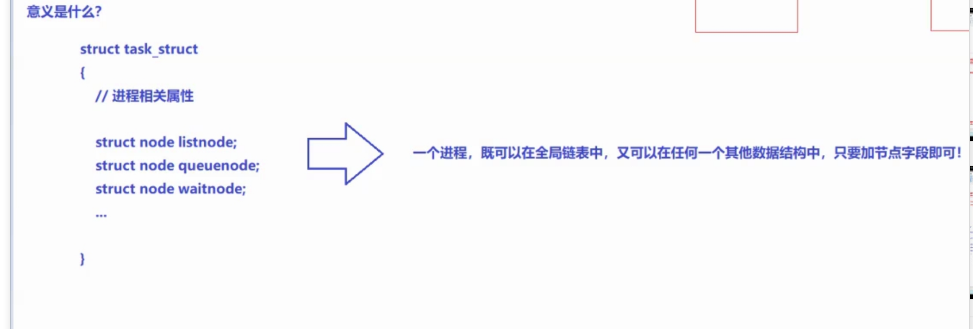
**图二**

**Linux所用的双向链表与我们所学的双向链表不同，他是如上图的结构，图一的结构位我们平常所学的结构，这种结构过于死板，不能实现在各个队列灵活切换进程。**

**图二的结构优点**

**内核链表的设计精髓：link结构体并不指向下一个task\_sturct节点而是指向他的link结构体。**

**意义是什么？**

****

**一个进程既可以在全局链表中，他又可以在不脱离全局链表的情况下，把他放入另一个链表里，因此同一个进程既可以在系统队列里又可以在运行队列里**

**进程在操作系统当中，无论你在哪都要在全局链表中维护**

**原理是什么？**

**如何通过node link访问其他节点**

**1. ​​容器化链表（container\_of）​​**

**Linux 使用通用双向链表结构 list\_head：**

**struct list\_head {**

**struct list\_head \*next, \*prev;**

**};**

* **​​妙用​​：通过 container\_of宏反向定位宿主结构**

**（如通过 run\_list地址计算出所在 task\_struct的地址）**

* **​​优势​​：一套链表操作函数可管理所有队列**

**在 Linux 内核中，container\_of宏是链表实现的核心魔法，它允许通过链表节点的地址​​反向计算出包含该节点的宿主结构的地址​​。这种设计称为​​内嵌链表​​（Intrusive Linked List）**

**三、计算步骤详解（通过实例说明）**

**假设：**

* **某个 task\_struct实例 my\_task的内存地址为 0x1000**
* **run\_list在 task\_struct内部的偏移量为 0x200字节**

**步骤 1: 确定链表节点地址 (ptr)**

**当我们在调度队列中遍历链表时，获得的指针是：**

**ptr = &my\_task->run\_list = 0x1000 + 0x200 = 0x1200**

**步骤 2: 计算成员偏移量 (offsetof)**

**编译器提供的 offsetof宏：**

***// 计算 run\_list 在 task\_struct 中的偏移量***

**size\_t offset = offsetof(struct task\_struct, run\_list); *// 返回 0x200***

**步骤 3: 反向计算宿主地址**

**host\_address = (char \*)ptr - offset**

**= 0x1200 - 0x200**

**= 0x1000 *// 即 my\_task 的地址***

**图解内存布局：**

**内存地址结构体对象地址最低与结构体第变量一个地址相同**

**内存地址 内容 说明**

**------------------------------------------------------------------**

**0x1000 | task\_struct 起始地址 | <-- 我们想得到的宿主结构地址**

**| ... |**

**| pid 等字段... |**

**0x1200 | list\_head.run\_list | <-- 我们已知的链表节点地址 (ptr)**

**| .next = ... |**

**| .prev = ... |**

**0x1210 | ... 其他字段 |**

**四、关键机制分析**

**1. offsetof的工作原理**

**编译器在编译时计算成员偏移量：**

**#define offsetof(type, member) ((size\_t)&((type \*)0)->member)**

* **(type \*)0：将地址 0 强制转换为 type\*类型**
* **&((type \*)0)->member：获取成员在结构体中的地址（此时在地址 0 上）**
* **计算结果就是成员距离结构体起始地址的字节偏移量**

**2. 类型安全机制**

**const typeof(((type \*)0)->member) \*\_\_mptr = (ptr);**

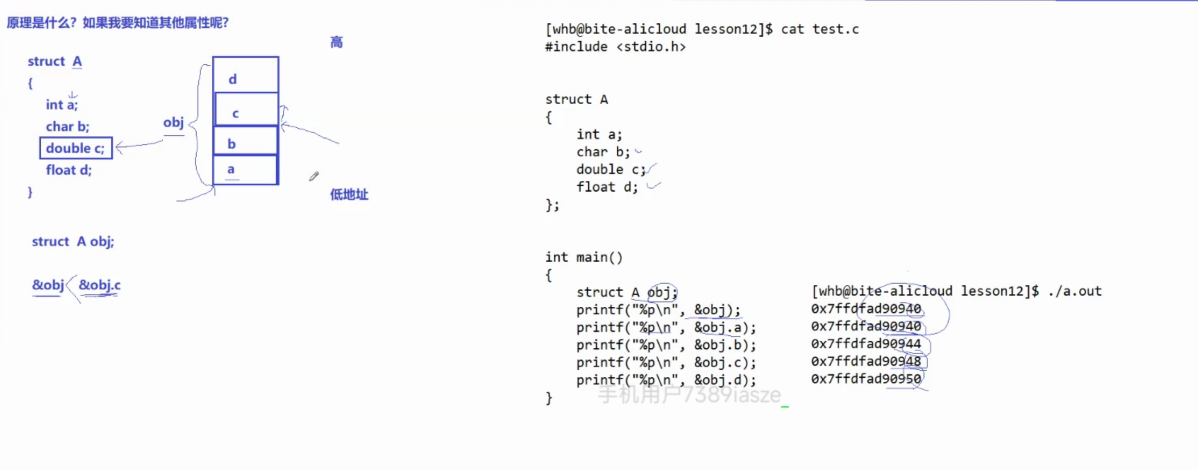
* **typeof获取成员的类型（确保 ptr与成员类型匹配）**
* **提供编译时的类型检查，防止错误操作**

**3. 字符指针转换**

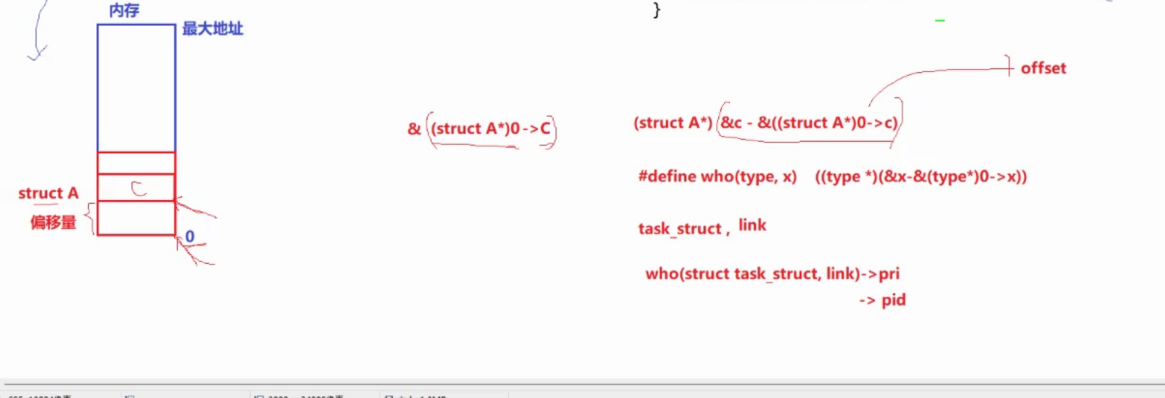
**(char \*)\_\_mptr**

* **转换为 char\*指针是为了进行​​字节级别的地址计算​​**
* **确保地址减法按字节计算而不是按结构体大小计算**

**Linux把进程维护起来包括他队列的设计都是用双链表维护的，但是该双链表yu1我们所学不同**

****

**结构体内内存地址图**

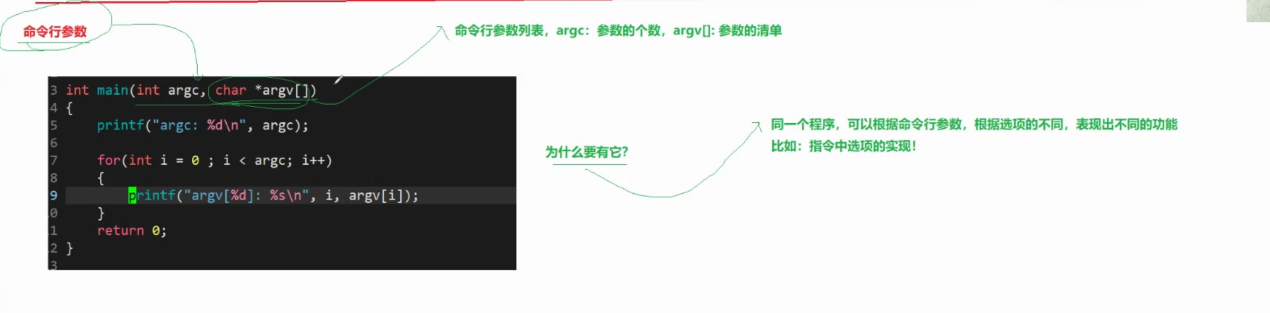
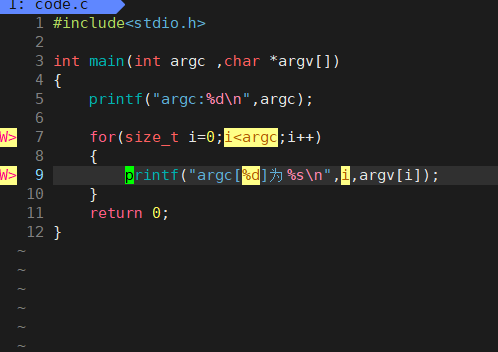
****

**如果计算结构体内变量的偏移量**

**环境变量**

**命令行参数**

**7/17/2025**

**  
**

**之前我们就了解到linux命令是一个一个的程序，我们使用linux命令的时候会有很多选项，我们的程序是如何拿到这些选项的呢？**

**主要由于，我们的命令行参数最终会被我们的解析出来，然后传递给程序，程序会根据选项的不同来实现不同的功能**

****

**一个简易的命令程序如上图**

**命令行参数列表，argc:参数的个数，argv[]:参数的清单**

**为啥要有命令行参数列表？**

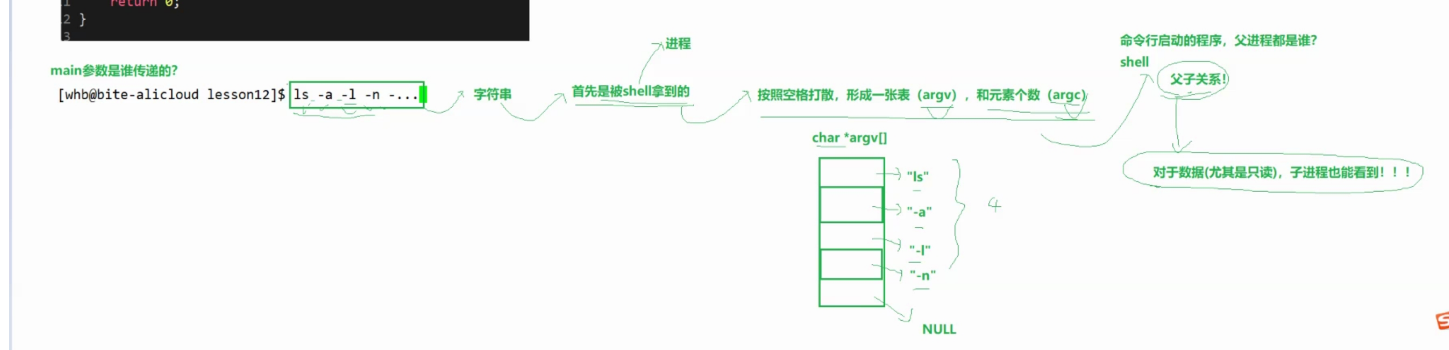
**同一个程序，可以根据命令行参数，根据选项都不同，表现出不同的功能比如：指令中选项的实现**

**Main参数是谁传递的？**

**命令行启动的程序，父进程都是谁？**

**命令行和启动的命令是父子关系**

**对于数据尤其是只读，子进程也能看到。**

****

**由于对于数据尤其是只读，子进程也能看到。因此子进程能够看到shell解析后的表（由命令字符串解析而成），子进程会将解析后的表和参数个数以参数的形式传入子程序，所以程序就读到了shell命令行解析之后的选项。**

**也就是说命令行参数的打散和形成是由shell来做的，然后我们后面执行的程序是子进程，他是shell以fork的形式创建出来的，表和参数只读，因此子进程也能读到，他会把对应argc,argv[], 传递给main函数。Main函数是你自己程序代码第一个被调用的函数入口函数。main函数谁调用，main函数是我们自己写程序的入口，但是系统调用我这个程序的时候不一定从main函数开始。**

**核心概念：谁才是真正的程序起点？​​**

**我们通常认为 main()或 WinMain()是程序的起点，但实际上这是不准确的。在你编写的 main()函数被调用*之前*，操作系统和 C 运行时库 (CRT - C Runtime Library) 已经默默做了大量的准备工作。CRTstartup()就是这个准备工作的主要执行者，是链接器设置的真正入口点。**

**CRTstartup()的本质和作用**

1. **​​程序入口点：​​ 当你双击一个 .exe文件时，操作系统的加载器 (Loader) 首先将程序的可执行文件加载到内存中。加载器随后跳转到的第一个地址，就是链接器 (Linker) 指定的“入口点” (Entry Point)。对于使用标准 Microsoft CRT 的 VC++ 程序，这个入口点通常命名为 mainCRTStartup（控制台程序）、wmainCRTStartup（宽字符控制台程序）、WinMainCRTStartup（GUI 程序）或 wWinMainCRTStartup（宽字符 GUI 程序）。​​它们本质上都是 CRTstartup()的具体实现变体，统称为 CRT 启动函数。​​**
2. **​​CRT 初始化：​​ CRT 启动函数的核心任务就是​​初始化 C/C++ 运行时环境​​，为你的 main()函数能正确运行搭建舞台。这包括：**
   * **​​初始化静态数据：​​ 设置全局变量、静态变量的内存空间（.data段初始化为初始值，.bss段初始化为零）。**
   * **​​构建参数列表：​​ 解析命令行参数 (argc, argv) 或 WinMain 的参数 (hInstance, hPrevInstance, lpCmdLine, nCmdShow)，并将它们准备好传递给 main()/ WinMain()。**
   * **​​初始化内存管理系统：​​ 设置 CRT 的堆管理器（malloc, free, new, delete等）。这包括创建初始化堆，设置内部内存结构，启用安全机制（如堆 Cookie、缓冲区安全检查 /GS）。**
   * **​​初始化 C++ 运行时：​​ 对于 C++ 程序：**
     + **调用全局静态对象的构造函数（在进入 main()*之前*）。**
     + **设置 std::cin, std::cout, std::cerr, std::clog等标准流对象（/SUBSYSTEM:CONSOLE）。**
     + **设置各种 C++ 库功能所需的环境（如异常处理框架、RTTI 等）。**
   * **​​初始化安全机制：​​ 设置缓冲区安全检查 (/GS), 安全 Cookie，调用 C 运行时安全检查函数。**
   * **​​调用初始化函数：​​ 执行由 #pragma init\_seg或其他机制指定的用户或库的初始化函数（在 main()之前运行）。**
   * **​​初始化浮点单元(FPU)：​​ 设置 FPU 的控制字和状态字。**
   * **​​初始化线程局部存储(TLS)：​​ 设置 TLS 数据。**
3. **​​调用主函数：​​ 在所有初始化工作完成后，启动函数才会 ​​调用 main()(或 WinMain(), wmain(), wWinMain())​​，并传递准备好的参数 (argc, argv或 hInstance, hPrevInstance, lpCmdLine, nCmdShow)。**
4. **​​主函数返回后的处理：​​ 当你的 main()函数执行完毕并返回时，控制权又交回给 CRT 启动函数。它会进行清理工作：**
   * **​​调用退出函数：​​ 调用通过 atexit()或 \_onexit()注册的函数（按注册顺序的​​相反​​顺序调用）。**
   * **​​销毁全局/静态对象：​​ 对于 C++ 程序，调用全局静态对象的析构函数（在调用完 atexit注册的函数*之后*）。**
   * **​​刷新和关闭标准 I/O：​​ 确保 std::cout等流的数据被刷新（写入）。**
   * **​​终止 CRT：​​ 执行 CRT 内部的最终清理工作。**
   * **​​调用系统退出：​​ 最终，调用操作系统提供的 ExitProcessAPI 或其等价物，将 main()的返回值 (int或 DWORD) 作为进程退出代码传递给操作系统，结束整个进程。**

**编译器**

**操作系统**

**加载器**

**不是相互割裂的，彼此是有关系的**

**加载器的重要性​​**

**加载器是将磁盘上“死”的可执行文件“复活”为内存中“活”的进程的核心桥梁。它默默无闻地完成了：**

1. **​​建立执行环境：​​ 创建进程、分配地址空间。**
2. **​​数据搬家：​​ 将程序的代码和数据高效（惰性加载）地映射到内存的正确位置。**
3. **​​连接纽带：​​ 解决动态库依赖，实现符号地址的绑定（重定位/GOT）。**
4. **​​初始化内存结构：​​ 设置栈和堆的初始状态。**
5. **​​安全保障：​​ 实施 ASLR、DEP 等安全策略。**
6. **​​启动引擎：​​ 最终启动程序代码执行（通过移交控制权给入口点）。**

**环境变量**

****

**形式为key=value**

**以key value形式构建的，具有全局属性的变量，叫做全局变量**

**为啥这样叫？？**

**当子进程启动时，父进程的环境变量表会传入子进程，每一个进程都有一个环境变量表，继承于他的父进程。**

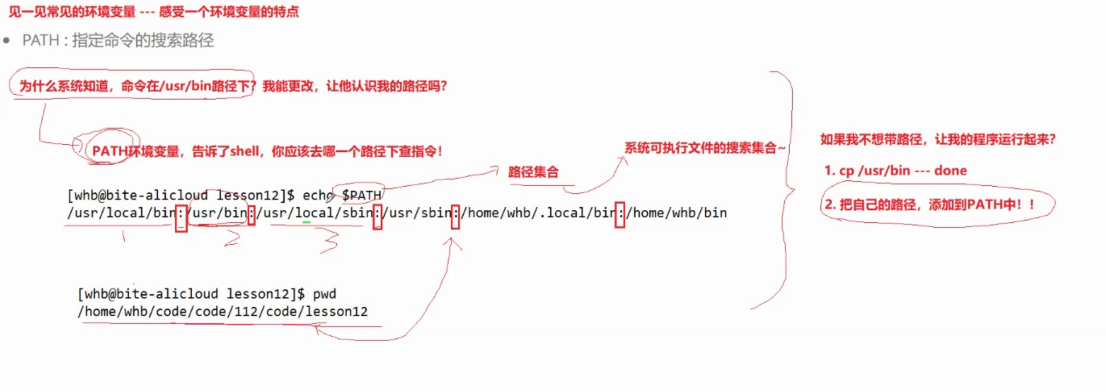
**执行任何一个程序首先要找到他，因为程序代码要加载形成进程。Ls pwd不用带路径是因为系统本来就能找到他，所以我们的code要带./就是告诉系统帮我执行这个路径下的code程序。**

**为啥系统知道命令在/usr/bin目录下。我能更改，让他认识我的路径吗？  
查找命令是·SHELL帮我们做的，PATH告诉了shell,你应该去哪个路径下查指令，根本原因不是shell认usr/bin而是shell认这个环境变量。**

**如果你想查环境变量 env**

**看单个环境变量**

**Echo $PATH**

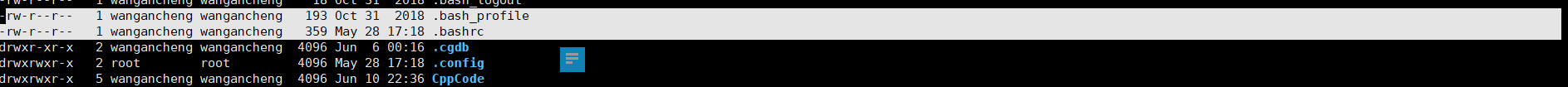
****

**通过上图我们就能认识到有因为code的路径并没有在系统可执行文件的搜索集合力因此不能直接用code来运行。**

**PATH=路径是覆盖式的写法不能这么写**

**环境变量已经被加载到了bash进程内部他是内存级的，也就是他已经被导入了一个进程的上下文里，不用担心环境变量改错了**

**为啥用户在登录linux系统时默认一定要处在自己的家目录吗？？**

**因为只有在自己的家目录下，他才能找到约定好的两个配置文件。**

**.bash\_profile中有环境变量**

**你在启动这个进程时，系统怎么知道你是谁，并且把你的uid写入到进程的pcb中？**

**Bash启动时会去加载与用户名相关的环境变量，环境变量里有**

**所以系统是怎么设置的呢？**

**首先系统知道我们的用户名是谁，系统一旦发现我们的用户名不是root，就会在我们的用户名后加/home,然后把环境变量一设置，然后chdir将bash的路径改成你当前的工作路径，所以当前用户就到了家目录下。**

**所以环境变量home记录的是当前用户所对应的家目录环境变量**

**Echo $**

**Echo $ PWD ----保存当前进程工作路径**

**为啥要保存当前进程工作路径呢？**