BFS简介Linux系统提速的必修课Linux认证考试 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/644/2021_2022_BFS_E7_AE_80_E4_BB_8BL_c103_644850.htm 像以往一样，依然在不断编译新版 Linux kernel 内核Linux 系统提速的必修课。和 Linux kernel BFS 的相遇充满了巧合下的必然。现在看来，BFS Kernel 是 Linux 在半年内给我的最大惊喜系统像电视购物主持人一样充满了力量和激情!而且是人能感觉得到的快!特以此文献给系统编译狂人，桌面提速狂 Linux 控。向所有 Linux 桌面用户力顶 BFS。 像以往一样，依然在不断编译新版 Linux kernel 内核Linux 系统提速的必修课。和 Linux kernel BFS 的相遇充满了巧合下的必然。现在看来，BFS Kernel 是 Linux 在半年内给我的最大惊喜系统像电视购物主持人一样充满了力量和激情!而且是人能感觉得到的快!特以此文献给系统编译狂人，桌面提速狂 Linux 控。向所有 Linux 桌面用户力顶 BFS。 最先在 Kindle 上看 xkcd 漫画，有漫画如是： A: 经过某些人千百年的努力，最新的 Linux 补丁支持 4096 个 CPU 的电脑了!原来只能支持 1024 个! B: 全屏 Flash 视频卡不卡啊? A: 卡。不过谁他丫的看视频啊? 而关于 BFS 的消息是最先在 Linux Magazine 上看到的.不久之后 G1 Android 手机 ROM 修改大神 CM 开始在他的测试版 CyanogenMod 使用 BFS 作为 kernel 的 Scheduler，试用之后发现手机系统速度明显加快。 用手滑动左右翻屏就像 Opera 下滚动网页那么平滑，搞得屏幕覆膜上多了好多指纹印。心痒已久，恰逢 Linux kernel 2.6.31 新版正式发布，打上 BFS Patch 编译，重启。神一样的提速再次出现在我 4 年高龄的笔记本电脑上，注入了鸡血的 KDE4 让人无

比兴奋。快!快!快!所以，BFS 是什么? 要知道 BFS 是什么最好先了解一下它的作者，传说中的澳洲猛士 CK。 CK，Con Kolivas，男， 澳大利亚中年男子，资深内核 hacker。众所周知，Linux Kernel 是聚集了一帮天才蠢才和暴君怪胎的地方，CK 貌似最适合这种地方的人。是真的貌似，一张电影里面典型高智商通缉犯的脸。 几年前编译 Linux kernel，ck 补丁集就是系统提速的代名词。当时编译内核的三部曲是下 kernel 源码，打上 ck 补丁集，编译安装。后来上游代码将 ck 补丁集稳定的部分不断吸收，它的影响力也渐渐消失。 CK 本身对任务调度有很深的造诣，他聪明而经典地实现了 fair scheduling，而实现模式被 Igor 借鉴改进最终写出了现在 kernel 用的进程调度管理器 CFS (Completely Fair Scheduler)。不得不顺便介绍一下任务调度。Kernel 的进程调度主要是将 CPU 资源分配给各种驱动、进程等等。你可能听说过，一般人的大脑使用率不足 20% 这种科学或者伪科学言论。但事实是，你电脑上的 CPU 从来就没有真正被 100% 的利用过(别跟我说你在资源管理器里面看到过 CPU 100%，我还见过 101% 呢)。如何将各种运算任务一刻不停又有条不紊的塞给 CPU 处理是一门严肃的科学，绝不是电视购物导购能解决的问题。一次塞的运算量少了，CPU 闲着，运算时间增长，电脑慢了.而一次塞的运算多了，CPU 忙不过来，运算又要在门口排队，电脑也慢了。进程调度主要是用算法解决这个问题，而现在 Linux Kernel 用的 CFS 据说非常经典，在不同情况下都可达到相当高的 CPU 利用率。而现用 CFS 也是在 2.6.23 才加入的，取代原来 O(1)，直接将 Linux 桌面速度从 XX 时代带入了 XX N 时代。 两年前，CK 淡出了内核开发，忽然从江湖中蒸

发。几周前，CK 重出江湖，两年磨一剑，带来了 BFS，全称 Brain Fuck Scheduler (只认识中间那个单词的请参考谷歌翻译)，声称专为低端硬件设计(我的理解是不超过 10 个 CPU 的电脑电视手机游戏机都算低端机)，说白了就是比 Kernel 默认要更加山崩地裂海枯石烂房价上涨油价飞升的快。BFS 为什么叫这个名字?为了中文用户，不能三个词让他们一个也不懂吧? 好吧，这名字有点不雅，不过算是直爽。对了，据说 CK 也是看到上面我提到的漫画才开始剑走偏锋。真正有几个人用有上千 CPU 的电脑呢?为什么要为这种扩展性牺牲桌面性能。BFS 就在其间做了取舍，仅仅支持最多 16 个 CPU，把问题外沿做小，让算法更简单精悍高效。作为原理来讲，这足够解释速度的来源。对于其它废问题， CK 专门写了一个 FAQ。在可以预见的将来，BFS 也不会进入 mainline kernel，说白了是取向问题。 关键问题是怎么用? 下 2.6.31 的 kernel 源代码，如果你不知道在哪里下的话就不必往下看了，在当前历史时期您还是搞不定的。再去：http://ck.kolivas.org/patches/bfs/ 下第一个 patch，现在是 2.6.31 开头的，表示适用该版本。解压内核源码，打上 patch，配置以后编译安装。现在 BFS 还在测试期，没有完全成熟，但已经相当可用。编译的时候有什么需要配置的?不需要，Scheduler 这东西太底层了，打上补丁就把原来的 CFS 替换掉了，没什么选项给你选。如果你非要问的话，不就图个快么，记着把配置弄到 1000Hz，开 preempt，禁掉 dynamic ticks。编译重启不用说了，我可以酷酷的扔下一个 have fun 然后去玩 Mac 了，反正你机器启动不了不要找我。虽然我纯净 kernel 单加 BFS Patch 编译成功启动没问题，依然有一位倒霉

的推油编译以后不知道怎么折腾的无法启动。可另外被我忽悠成功的推友们反应一致：“ 快!人能感觉得到的快! ”到底值不值得上手，有没有评测? 这是某些不够剽悍的读者会挣扎到最后的问题。BFS 原理上讲，机器配置越低，感受会越明显。如果你非要评测的话，Phoronix 这个专业的 Linux 测评狂网站也出了一份。我可以提前剧透结论，区别都很小，BFS 胜出绝大部分测试，然而优势不明显。我只是补充一下绝大多数折腾过的人的感受 快 !人能感觉到的快! struct sock *next. struct sock *prev. /* Doubly linked chain.. */ struct sock *pair. struct sk_buff * volatile send_head. struct sk_buff * volatile send_tail. struct sk_buff_head back_log. struct sk_buff *partial. struct timer_list partial_timer. long retransmits. struct sk_buff_head write_queue, receive_queue. struct proto *prot. struct wait_queue **sleep. unsigned long daddr. unsigned long saddr. unsigned short max_unacked. unsigned short window. unsigned short bytes_rcv. /* mss is min(mtu, max_window) */ unsigned short mtu. /* mss negotiated in the syns */ volatile unsigned short mss. /* current eff. mss - can change */ volatile unsigned short user_mss. /* mss requested by user in ioctl */ volatile unsigned short max_window. unsigned long window_clamp. unsigned short num. volatile unsigned short cong_window. volatile unsigned short cong_count. volatile unsigned short ssthresh. volatile unsigned short packets_out. volatile unsigned short shutdown. volatile unsigned long rtt. volatile unsigned long mdev. volatile unsigned long rto. /* currently backoff isnt used, but Im maintaining it in case * we want to go back to a backoff formula that needs it */ volatile unsigned short backoff.

volatile short err. unsigned char protocol. volatile unsigned char state. volatile unsigned char ack_backlog. unsigned char max_ack_backlog. unsigned char priority. unsigned char debug. unsigned short rcvbuf. unsigned short sndbuf. unsigned short type. unsigned char localroute. /* Route locally only */ #ifdef CONFIG_IPX ipx_address ipx_dest_addr. ipx_interface *ipx_intrfc. unsigned short ipx_port. unsigned short ipx_type. #endif#ifdef CONFIG_AX25 /* Really we want to add a per protocol private area */ ax25_address ax25_source_addr,ax25_dest_addr. struct sk_buff *volatile ax25_retxq[8]. char ax25_state,ax25_vs,ax25_vr,ax25_lastrxnr,ax25_lasttxnr. char ax25_condition. char ax25_retxcnt. char ax25_xx. char ax25_retxqi. char ax25_rrtimer. char ax25_timer. unsigned char ax25_n2. unsigned short ax25_t1,ax25_t2,ax25_t3. ax25_digi *ax25_digipeat. #endif #ifdef CONFIG_ATALK struct atalk_sock at. #endif /* IP private area or will be eventually */ int ip_ttl. /* TTL setting */ int ip_tos. /* TOS */ struct tcphdr dummy_th. struct timer_list keepalive_timer. /* TCP keepalive hack */ struct timer_list retransmit_timer. /* TCP retransmit timer */ struct timer_list ack_timer. /* TCP delayed ack timer */ int ip_xmit_timeout. /* Why the timeout is running */ #ifdef CONFIG_IP_MULTICAST int ip_mc_ttl. /* Multicasting TTL */ int ip_mc_loop. /* Loopback (not implemented yet) */ char ip_mc_name[MAX_ADDR_LEN]. /* Multicast device name */ struct ip_mc_socklist *ip_mc_list. /* Group array */ #endif /* This part is used for the timeout functions (timer.c). */ int timeout. /* What are we waiting for? */ struct

timer_list timer. /* This is the TIME_WAIT/receive timer when we are doing IP */ struct timeval stamp. /* identd */ struct socket *socket. /* Callbacks */ void (*state_change)(struct sock *sk). void (*data_ready)(struct sock *sk,int bytes). void (*write_space)(struct sock *sk). void (*error_report)(struct sock *sk). }. 2结构用途 TCP协议的核心结构，只应用于TCP协议。 3语句注释 3.1 成员注释opt：IP选项缓存于此处。 wmem_alloc：当前写缓冲区大小，该值不可大于系统规定的最大值。 rmem_alloc：当前读缓冲区大小，该值不可大于系统规定最大值。 write_seq：表示应用程序下一次写数据时所对应的第一个字节的序列号，write_queue队列的序列号。 sent_seq：表示本地将要发送的下一个数据包中第一个字节对应的序列号。 acked_seq：表示本地希望从远端接收的下一个数据的序列号。 copied_seq：应用程序有待读取但尚未读取的数据的第一个字节的序列号。 rcv_ack_seq：表示目前本地接收到的远端对本地发送数据的应答序列号，表示本地已经发送的小于此序列号的所有数据已被远端成功接收。 window_seq：sent_seq值加上远端窗口的大小，本地将要发送的数据包的最后一个字节的序列号不可大于此值。