# 第一章概述

## 1.1 linux内核双链表结构

### 1.1.1原理

1）内核的链表的优点在于，如果仅仅在头部做插入与删除 就是一个栈，如果头部插入，尾部删除就是一个队列。也可以退化一个二叉树。

2）函数解释

### 1.1.1.2.1

//链表结构体

struct list\_head {

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

};

//初始化一个链表

static inline void init\_list\_head(struct list\_head \*list)

{

list->next = list;

list->prev = list;

}

//添加一个节点进入链表—头部插入--

static void \_\_list\_add(struct list\_head \*new, struct list\_head \*prev ,struct list\_head \*next)

{

next->prev = new;

new->next = next;

new->prev = prev;

prev->next = new;

}



//遍历所有的结构体

#define list\_for\_each(pos ,head) \

for(pos = (head)->next ; pos != (head) ;pos = pos->next)

//求出在结构体中的偏移量

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

//计算结构体的首地址

#define container\_of(ptr, type, member) ({ \

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

(type \*)( (char \*)\_\_mptr - offsetof(type,member) );})

//计算结构体的首地址

#define list\_entry(ptr, type, member) \

container\_of(ptr, type, member)

//list\_for\_each\_entry = list\_for\_each + list\_entry

#define list\_for\_each\_entry(pos, head, member) \

for (pos = list\_entry((head)->next, typeof(\*pos), member); \

&pos->member != (head); \

pos = list\_entry(pos->member.next, typeof(\*pos), member))

//对于list\_for\_each\_entry的安全操作 (n和pos属于同一种类型)

#define list\_for\_each\_entry\_safe(pos, n, head, member) \

for (pos = list\_entry((head)->next, typeof(\*pos), member), \

n = list\_entry(pos->member.next, typeof(\*pos), member); \

&pos->member != (head); \

pos = n, n = list\_entry(n->member.next, typeof(\*n), member))

### 1.1.2 实验

## 1.2 linux内核中的哈希表

### 1.2.1原理

### 1.2.2 实验

## 总结:

# 第二章进程

## 2.1进程管理

### 1.2.1进程结构体task\_struct

<https://blog.csdn.net/jelly_9/article/details/78735967>

位置：include/linux/sched.h

*Struct task\_struct*

*{*

*/\**

*1. state: 进程执行时，它会根据具体情况改变状态。进程状态是进程调度和对换的依据。Linux中的进程主要有如下状态:*

*#define TASK\_RUNNING 0*

*1) TASK\_RUNNING: 可运行*

*处于这种状态的进程，只有两种状态:*

*1.1) 正在运行*

*正在运行的进程就是当前进程(由current所指向的进程)*

*1.2) 正准备运行*

*准备运行的进程只要得到CPU就可以立即投入运行，CPU是这些进程唯一等待的系统资源，系统中有一个运行队列(run\_queue)，用来容纳所有处于可运行状态的进程，调度程序执行时，从中选择一个进程投入运行*

*#define TASK\_INTERRUPTIBLE 1*

*2) TASK\_INTERRUPTIBLE: 可中断的等待状态，是针对等待某事件或其他资源的睡眠进程设置的，在内核发送信号给该进程表明事件已经发生时，进程状态变为TASK\_RUNNING，它只要调度器选中该进程即可恢复执行*

*#define TASK\_UNINTERRUPTIBLE 2*

*3) TASK\_UNINTERRUPTIBLE: 不可中断的等待状态*

*处于该状态的进程正在等待某个事件(event)或某个资源，它肯定位于系统中的某个等待队列(wait\_queue)中，处于不可中断等待态的进程是因为硬件环境不能满足而等待，例如等待特定的系统资源，它任何情况下都不能被打断，只能用特定的方式来唤醒它，例如唤醒函数wake\_up()等*

*它们不能由外部信号唤醒，只能由内核亲自唤醒*

*4) TASK\_ZOMBIE: 僵死*

*进程虽然已经终止，但由于某种原因，父进程还没有执行wait()系统调用，终止进程的信息也还没有回收。顾名思义，处于该状态的进程就是死进程，这种进程实际上是系统中的垃圾，必须进行相应处理以释放其占用的资源。*

*5) TASK\_STOPPED: 暂停*

*此时的进程暂时停止运行来接受某种特殊处理。通常当进程接收到SIGSTOP、SIGTSTP、SIGTTIN或 SIGTTOU信号后就处于这种状态。例如，正接受调试的进程就处于这种状态*

*6) TASK\_TRACED*

*从本质上来说，这属于TASK\_STOPPED状态，用于从停止的进程中，将当前被调试的进程与常规的进程区分开来*

*7) TASK\_DEAD*

*父进程wait系统调用发出后，当子进程退出时，父进程负责回收子进程的全部资源，子进程进入TASK\_DEAD状态*

*8) TASK\_SWAPPING: 换入/换出*

*\*/*

*volatile long state;*

*/\**

*2. stack*

*进程内核栈，进程通过alloc\_thread\_info函数分配它的内核栈，通过free\_thread\_info函数释放所分配的内核栈*

*\*/*

*void \*stack;*

*/\**

*3. usage*

*进程描述符使用计数，被置为2时，表示进程描述符正在被使用而且其相应的进程处于活动状态*

*\*/*

*atomic\_t usage;*

*/\**

*4. flags*

*flags是进程当前的状态标志(注意和运行状态区分)*

*1) #define PF\_ALIGNWARN 0x00000001: 显示内存地址未对齐警告*

*2) #define PF\_PTRACED 0x00000010: 标识是否是否调用了ptrace*

*3) #define PF\_TRACESYS 0x00000020: 跟踪系统调用*

*4) #define PF\_FORKNOEXEC 0x00000040: 已经完成fork，但还没有调用exec*

*5) #define PF\_SUPERPRIV 0x00000100: 使用超级用户(root)权限*

*6) #define PF\_DUMPCORE 0x00000200: dumped core*

*7) #define PF\_SIGNALED 0x00000400: 此进程由于其他进程发送相关信号而被杀死*

*8) #define PF\_STARTING 0x00000002: 当前进程正在被创建*

*9) #define PF\_EXITING 0x00000004: 当前进程正在关闭*

*10) #define PF\_USEDFPU 0x00100000: Process used the FPU this quantum(SMP only)*

*#define PF\_DTRACE 0x00200000: delayed trace (used on m68k)*

*\*/*

*unsigned int flags;*

*/\**

*5. ptrace*

*ptrace系统调用，成员ptrace被设置为0时表示不需要被跟踪，它的可能取值如下：*

*linux-2.6.38.8/include/linux/ptrace.h*

*1) #define PT\_PTRACED 0x00000001*

*2) #define PT\_DTRACE 0x00000002: delayed trace (used on m68k, i386)*

*3) #define PT\_TRACESYSGOOD 0x00000004*

*4) #define PT\_PTRACE\_CAP 0x00000008: ptracer can follow suid-exec*

*5) #define PT\_TRACE\_FORK 0x00000010*

*6) #define PT\_TRACE\_VFORK 0x00000020*

*7) #define PT\_TRACE\_CLONE 0x00000040*

*8) #define PT\_TRACE\_EXEC 0x00000080*

*9) #define PT\_TRACE\_VFORK\_DONE 0x00000100*

*10) #define PT\_TRACE\_EXIT 0x00000200*

*\*/*

*unsigned int ptrace;*

*unsigned long ptrace\_message;*

*siginfo\_t \*last\_siginfo;*

*/\**

*6. lock\_depth*

*用于表示获取大内核锁的次数，如果进程未获得过锁，则置为-1*

*\*/*

*int lock\_depth;*

*/\**

*7. oncpu*

*在SMP上帮助实现无加锁的进程切换(unlocked context switches)*

*\*/*

*#ifdef CONFIG\_SMP*

*#ifdef \_\_ARCH\_WANT\_UNLOCKED\_CTXSW*

*int oncpu;*

*#endif*

*#endif*

*/\**

*8. 进程调度*

*1) prio: 调度器考虑的优先级保存在prio，由于在某些情况下内核需要暂时提高进程的优先级，因此需要第三个成员来表示(除了static\_prio、normal\_prio之外)，由于这些改变不是持久的，因此静态(static\_prio)和普通(normal\_prio)优先级不受影响*

*2) static\_prio: 用于保存进程的"静态优先级"，静态优先级是进程"启动"时分配的优先级，它可以用nice、sched\_setscheduler系统调用修改，否则在进程运行期间会一直保持恒定*

*3) normal\_prio: 表示基于进程的"静态优先级"和"调度策略"计算出的优先级，因此，即使普通进程和实时进程具有相同的静态优先级(static\_prio)，其普通优先级(normal\_prio)也是不同的。进程分支时(fork)，新创建的子进程会集成普通优先级*

*\*/*

*int prio, static\_prio, normal\_prio;*

*/\**

*4) rt\_priority: 表示实时进程的优先级，需要明白的是，"实时进程优先级"和"普通进程优先级"有两个独立的范畴，实时进程即使是最低优先级也高于普通进程，最低的实时优先级为0，最高的优先级为99，值越大，表明优先级越高*

*\*/*

*unsigned int rt\_priority;*

*/\**

*5) sched\_class: 该进程所属的调度类，目前内核中有实现以下四种：*

*5.1) static const struct sched\_class fair\_sched\_class;*

*5.2) static const struct sched\_class rt\_sched\_class;*

*5.3) static const struct sched\_class idle\_sched\_class;*

*5.4) static const struct sched\_class stop\_sched\_class;*

*\*/*

*const struct sched\_class \*sched\_class;*

*/\**

*6) se: 用于普通进程的调用实体*

*调度器不限于调度进程，还可以处理更大的实体，这可以实现"组调度"，可用的CPU时间可以首先在一般的进程组(例如所有进程可以按所有者分组)之间分配，接下来分配的时间在组内再次分配*

*这种一般性要求调度器不直接操作进程，而是处理"可调度实体"，一个实体有sched\_entity的一个实例标识*

*在最简单的情况下，调度在各个进程上执行，由于调度器设计为处理可调度的实体，在调度器看来各个进程也必须也像这样的实体，因此se在task\_struct中内嵌了一个sched\_entity实例，调度器可据此操作各个task\_struct*

*\*/*

*struct sched\_entity se;*

*/\**

*7) rt: 用于实时进程的调用实体*

*\*/*

*struct sched\_rt\_entity rt;*

*#ifdef CONFIG\_PREEMPT\_NOTIFIERS*

*/\**

*9. preempt\_notifier*

*preempt\_notifiers结构体链表*

*\*/*

*struct hlist\_head preempt\_notifiers;*

*#endif*

*/\**

*10. fpu\_counter*

*FPU使用计数*

*\*/*

*unsigned char fpu\_counter;*

*#ifdef CONFIG\_BLK\_DEV\_IO\_TRACE*

*/\**

*11. btrace\_seq*

*blktrace是一个针对Linux内核中块设备I/O层的跟踪工具*

*\*/*

*unsigned int btrace\_seq;*

*#endif*

*/\**

*12. policy*

*policy表示进程的调度策略，目前主要有以下五种：*

*1) #define SCHED\_NORMAL 0: 用于普通进程，它们通过完全公平调度器来处理*

*2) #define SCHED\_FIFO 1: 先来先服务调度，由实时调度类处理*

*3) #define SCHED\_RR 2: 时间片轮转调度，由实时调度类处理*

*4) #define SCHED\_BATCH 3: 用于非交互、CPU使用密集的批处理进程，通过完全公平调度器来处理，调度决策对此类进程给与"冷处理"，它们绝不会抢占CFS调度器处理的另一个进程，因此不会干扰交互式进程，如果不打算用nice降低进程的静态优先级，同时又不希望该进程影响系统的交互性，最适合用该调度策略*

*5) #define SCHED\_IDLE 5: 可用于次要的进程，其相对权重总是最小的，也通过完全公平调度器来处理。要注意的是，SCHED\_IDLE不负责调度空闲进程，空闲进程由内核提供单独的机制来处理*

*只有root用户能通过sched\_setscheduler()系统调用来改变调度策略*

*\*/*

*unsigned int policy;*

*/\**

*13. cpus\_allowed*

*cpus\_allowed是一个位域，在多处理器系统上使用，用于控制进程可以在哪里处理器上运行*

*\*/*

*cpumask\_t cpus\_allowed;*

*/\**

*14. RCU同步原语*

*\*/*

*#ifdef CONFIG\_TREE\_PREEMPT\_RCU*

*int rcu\_read\_lock\_nesting;*

*char rcu\_read\_unlock\_special;*

*struct rcu\_node \*rcu\_blocked\_node;*

*struct list\_head rcu\_node\_entry;*

*#endif /\* #ifdef CONFIG\_TREE\_PREEMPT\_RCU \*/*

*#if defined(CONFIG\_SCHEDSTATS) || defined(CONFIG\_TASK\_DELAY\_ACCT)*

*/\**

*15. sched\_info*

*用于调度器统计进程的运行信息*

*\*/*

*struct sched\_info sched\_info;*

*#endif*

*/\**

*16. tasks*

*通过list\_head将当前进程的task\_struct串联进内核的进程列表中，构建；linux进程链表*

*\*/*

*struct list\_head tasks;*

*/\**

*17. pushable\_tasks*

*limit pushing to one attempt*

*\*/*

*struct plist\_node pushable\_tasks;*

*/\**

*18. 进程地址空间*

*1) mm: 指向进程所拥有的内存描述符*

*2) active\_mm: active\_mm指向进程运行时所使用的内存描述符*

*对于普通进程而言，这两个指针变量的值相同。但是，内核线程不拥有任何内存描述符，所以它们的mm成员总是为NULL。当内核线程得以运行时，它的active\_mm成员被初始化为前一个运行进程的active\_mm值*

*\*/*

*struct mm\_struct \*mm, \*active\_mm;*

*/\**

*19. exit\_state*

*进程退出状态码*

*\*/*

*int exit\_state;*

*/\**

*20. 判断标志*

*1) exit\_code*

*exit\_code用于设置进程的终止代号，这个值要么是\_exit()或exit\_group()系统调用参数(正常终止)，要么是由内核提供的一个错误代号(异常终止)*

*2) exit\_signal*

*exit\_signal被置为-1时表示是某个线程组中的一员。只有当线程组的最后一个成员终止时，才会产生一个信号，以通知线程组的领头进程的父进程*

*\*/*

*int exit\_code, exit\_signal;*

*/\**

*3) pdeath\_signal*

*pdeath\_signal用于判断父进程终止时发送信号*

*\*/*

*int pdeath\_signal;*

*/\**

*4) personality用于处理不同的ABI，它的可能取值如下：*

*enum*

*{*

*PER\_LINUX = 0x0000,*

*PER\_LINUX\_32BIT = 0x0000 | ADDR\_LIMIT\_32BIT,*

*PER\_LINUX\_FDPIC = 0x0000 | FDPIC\_FUNCPTRS,*

*PER\_SVR4 = 0x0001 | STICKY\_TIMEOUTS | MMAP\_PAGE\_ZERO,*

*PER\_SVR3 = 0x0002 | STICKY\_TIMEOUTS | SHORT\_INODE,*

*PER\_SCOSVR3 = 0x0003 | STICKY\_TIMEOUTS |*

*WHOLE\_SECONDS | SHORT\_INODE,*

*PER\_OSR5 = 0x0003 | STICKY\_TIMEOUTS | WHOLE\_SECONDS,*

*PER\_WYSEV386 = 0x0004 | STICKY\_TIMEOUTS | SHORT\_INODE,*

*PER\_ISCR4 = 0x0005 | STICKY\_TIMEOUTS,*

*PER\_BSD = 0x0006,*

*PER\_SUNOS = 0x0006 | STICKY\_TIMEOUTS,*

*PER\_XENIX = 0x0007 | STICKY\_TIMEOUTS | SHORT\_INODE,*

*PER\_LINUX32 = 0x0008,*

*PER\_LINUX32\_3GB = 0x0008 | ADDR\_LIMIT\_3GB,*

*PER\_IRIX32 = 0x0009 | STICKY\_TIMEOUTS,*

*PER\_IRIXN32 = 0x000a | STICKY\_TIMEOUTS,*

*PER\_IRIX64 = 0x000b | STICKY\_TIMEOUTS,*

*PER\_RISCOS = 0x000c,*

*PER\_SOLARIS = 0x000d | STICKY\_TIMEOUTS,*

*PER\_UW7 = 0x000e | STICKY\_TIMEOUTS | MMAP\_PAGE\_ZERO,*

*PER\_OSF4 = 0x000f,*

*PER\_HPUX = 0x0010,*

*PER\_MASK = 0x00ff,*

*};*

*\*/*

*unsigned int personality;*

*/\**

*5) did\_exec*

*did\_exec用于记录进程代码是否被execve()函数所执行*

*\*/*

*unsigned did\_exec:1;*

*/\**

*6) in\_execve*

*in\_execve用于通知LSM是否被do\_execve()函数所调用*

*\*/*

*unsigned in\_execve:1;*

*/\**

*7) in\_iowait*

*in\_iowait用于判断是否进行iowait计数*

*\*/*

*unsigned in\_iowait:1;*

*/\**

*8) sched\_reset\_on\_fork*

*sched\_reset\_on\_fork用于判断是否恢复默认的优先级或调度策略*

*\*/*

*unsigned sched\_reset\_on\_fork:1;*

*/\**

*21. 进程标识符(PID)*

*在CONFIG\_BASE\_SMALL配置为0的情况下，PID的取值范围是0到32767，即系统中的进程数最大为32768个*

*#define PID\_MAX\_DEFAULT (CONFIG\_BASE\_SMALL ? 0x1000 : 0x8000)*

*在Linux系统中，一个线程组中的所有线程使用和该线程组的领头线程(该组中的第一个轻量级进程)相同的PID，并被存放在tgid成员中。只有线程组的领头线程的pid成员才会被设置为与tgid相同的值。注意，getpid()系统调用*

*返回的是当前进程的tgid值而不是pid值。*

*\*/*

*pid\_t pid;*

*pid\_t tgid;*

*#ifdef CONFIG\_CC\_STACKPROTECTOR*

*/\**

*22. stack\_canary*

*防止内核堆栈溢出，在GCC编译内核时，需要加上-fstack-protector选项*

*\*/*

*unsigned long stack\_canary;*

*#endif*

*/\**

*23. 表示进程亲属关系的成员*

*1) real\_parent: 指向其父进程，如果创建它的父进程不再存在，则指向PID为1的init进程*

*2) parent: 指向其父进程，当它终止时，必须向它的父进程发送信号。它的值通常与real\_parent相同*

*\*/*

*struct task\_struct \*real\_parent;*

*struct task\_struct \*parent;*

*/\**

*3) children: 表示链表的头部，链表中的所有元素都是它的子进程(子进程链表)*

*4) sibling: 用于把当前进程插入到兄弟链表中(连接到父进程的子进程链表(兄弟链表))*

*5) group\_leader: 指向其所在进程组的领头进程*

*\*/*

*struct list\_head children;*

*struct list\_head sibling;*

*struct task\_struct \*group\_leader;*

*struct list\_head ptraced;*

*struct list\_head ptrace\_entry;*

*struct bts\_context \*bts;*

*/\**

*24. pids*

*PID散列表和链表*

*\*/*

*struct pid\_link pids[PIDTYPE\_MAX];*

*/\**

*25. thread\_group*

*线程组中所有进程的链表*

*\*/*

*struct list\_head thread\_group;*

*/\**

*26. do\_fork函数*

*1) vfork\_done*

*在执行do\_fork()时，如果给定特别标志，则vfork\_done会指向一个特殊地址*

*2) set\_child\_tid、clear\_child\_tid*

*如果copy\_process函数的clone\_flags参数的值被置为CLONE\_CHILD\_SETTID或CLONE\_CHILD\_CLEARTID，则会把child\_tidptr参数的值分别复制到set\_child\_tid和clear\_child\_tid成员。这些标志说明必须改变子*

*进程用户态地址空间的child\_tidptr所指向的变量的值。*

*\*/*

*struct completion \*vfork\_done;*

*int \_\_user \*set\_child\_tid;*

*int \_\_user \*clear\_child\_tid;*

*/\**

*27. 记录进程的I/O计数(时间)*

*1) utime*

*用于记录进程在"用户态"下所经过的节拍数(定时器)*

*2) stime*

*用于记录进程在"内核态"下所经过的节拍数(定时器)*

*3) utimescaled*

*用于记录进程在"用户态"的运行时间，但它们以处理器的频率为刻度*

*4) stimescaled*

*用于记录进程在"内核态"的运行时间，但它们以处理器的频率为刻度*

*\*/*

*cputime\_t utime, stime, utimescaled, stimescaled;*

*/\**

*5) gtime*

*以节拍计数的虚拟机运行时间(guest time)*

*\*/*

*cputime\_t gtime;*

*/\**

*6) prev\_utime、prev\_stime是先前的运行时间*

*\*/*

*cputime\_t prev\_utime, prev\_stime;*

*/\**

*7) nvcsw*

*自愿(voluntary)上下文切换计数*

*8) nivcsw*

*非自愿(involuntary)上下文切换计数*

*\*/*

*unsigned long nvcsw, nivcsw;*

*/\**

*9) start\_time*

*进程创建时间*

*10) real\_start\_time*

*进程睡眠时间，还包含了进程睡眠时间，常用于/proc/pid/stat，*

*\*/*

*struct timespec start\_time;*

*struct timespec real\_start\_time;*

*/\**

*11) cputime\_expires*

*用来统计进程或进程组被跟踪的处理器时间，其中的三个成员对应着cpu\_timers[3]的三个链表*

*\*/*

*struct task\_cputime cputime\_expires;*

*struct list\_head cpu\_timers[3];*

*#ifdef CONFIG\_DETECT\_HUNG\_TASK*

*/\**

*12) last\_switch\_count*

*nvcsw和nivcsw的总和*

*\*/*

*unsigned long last\_switch\_count;*

*#endif*

*struct task\_io\_accounting ioac;*

*#if defined(CONFIG\_TASK\_XACCT)*

*u64 acct\_rss\_mem1;*

*u64 acct\_vm\_mem1;*

*cputime\_t acct\_timexpd;*

*#endif*

*/\**

*28. 缺页统计*

*\*/*

*unsigned long min\_flt, maj\_flt;*

*/\**

*29. 进程权能*

*\*/*

*const struct cred \*real\_cred;*

*const struct cred \*cred;*

*struct mutex cred\_guard\_mutex;*

*struct cred \*replacement\_session\_keyring;*

*/\**

*30. comm[TASK\_COMM\_LEN]*

*相应的程序名*

*\*/*

*char comm[TASK\_COMM\_LEN];*

*/\**

*31. 文件*

*1) fs*

*用来表示进程与文件系统的联系，包括当前目录和根目录*

*2) files*

*表示进程当前打开的文件*

*\*/*

*int link\_count, total\_link\_count;*

*struct fs\_struct \*fs;*

*struct files\_struct \*files;*

*#ifdef CONFIG\_SYSVIPC*

*/\**

*32. sysvsem*

*进程通信(SYSVIPC)*

*\*/*

*struct sysv\_sem sysvsem;*

*#endif*

*/\**

*33. 处理器特有数据*

*\*/*

*struct thread\_struct thread;*

*/\**

*34. nsproxy*

*命名空间*

*\*/*

*struct nsproxy \*nsproxy;*

*/\**

*35. 信号处理*

*1) signal: 指向进程的信号描述符*

*2) sighand: 指向进程的信号处理程序描述符*

*\*/*

*struct signal\_struct \*signal;*

*struct sighand\_struct \*sighand;*

*/\**

*3) blocked: 表示被阻塞信号的掩码*

*4) real\_blocked: 表示临时掩码*

*\*/*

*sigset\_t blocked, real\_blocked;*

*sigset\_t saved\_sigmask;*

*/\**

*5) pending: 存放私有挂起信号的数据结构*

*\*/*

*struct sigpending pending;*

*/\**

*6) sas\_ss\_sp: 信号处理程序备用堆栈的地址*

*7) sas\_ss\_size: 表示堆栈的大小*

*\*/*

*unsigned long sas\_ss\_sp;*

*size\_t sas\_ss\_size;*

*/\**

*8) notifier*

*设备驱动程序常用notifier指向的函数来阻塞进程的某些信号*

*9) otifier\_data*

*指的是notifier所指向的函数可能使用的数据。*

*10) otifier\_mask*

*标识这些信号的位掩码*

*\*/*

*int (\*notifier)(void \*priv);*

*void \*notifier\_data;*

*sigset\_t \*notifier\_mask;*

*/\**

*36. 进程审计*

*\*/*

*struct audit\_context \*audit\_context;*

*#ifdef CONFIG\_AUDITSYSCALL*

*uid\_t loginuid;*

*unsigned int sessionid;*

*#endif*

*/\**

*37. secure computing*

*\*/*

*seccomp\_t seccomp;*

*/\**

*38. 用于copy\_process函数使用CLONE\_PARENT标记时*

*\*/*

*u32 parent\_exec\_id;*

*u32 self\_exec\_id;*

*/\**

*39. alloc\_lock*

*用于保护资源分配或释放的自旋锁*

*\*/*

*spinlock\_t alloc\_lock;*

*/\**

*40. 中断*

*\*/*

*#ifdef CONFIG\_GENERIC\_HARDIRQS*

*struct irqaction \*irqaction;*

*#endif*

*#ifdef CONFIG\_TRACE\_IRQFLAGS*

*unsigned int irq\_events;*

*int hardirqs\_enabled;*

*unsigned long hardirq\_enable\_ip;*

*unsigned int hardirq\_enable\_event;*

*unsigned long hardirq\_disable\_ip;*

*unsigned int hardirq\_disable\_event;*

*int softirqs\_enabled;*

*unsigned long softirq\_disable\_ip;*

*unsigned int softirq\_disable\_event;*

*unsigned long softirq\_enable\_ip;*

*unsigned int softirq\_enable\_event;*

*int hardirq\_context;*

*int softirq\_context;*

*#endif*

*/\**

*41. pi\_lock*

*task\_rq\_lock函数所使用的锁*

*\*/*

*spinlock\_t pi\_lock;*

*#ifdef CONFIG\_RT\_MUTEXES*

*/\**

*42. 基于PI协议的等待互斥锁，其中PI指的是priority inheritance/9优先级继承)*

*\*/*

*struct plist\_head pi\_waiters;*

*struct rt\_mutex\_waiter \*pi\_blocked\_on;*

*#endif*

*#ifdef CONFIG\_DEBUG\_MUTEXES*

*/\**

*43. blocked\_on*

*死锁检测*

*\*/*

*struct mutex\_waiter \*blocked\_on;*

*#endif*

*/\**

*44. lockdep，*

*\*/*

*#ifdef CONFIG\_LOCKDEP*

*# define MAX\_LOCK\_DEPTH 48UL*

*u64 curr\_chain\_key;*

*int lockdep\_depth;*

*unsigned int lockdep\_recursion;*

*struct held\_lock held\_locks[MAX\_LOCK\_DEPTH];*

*gfp\_t lockdep\_reclaim\_gfp;*

*#endif*

*/\**

*45. journal\_info*

*JFS文件系统*

*\*/*

*void \*journal\_info;*

*/\**

*46. 块设备链表*

*\*/*

*struct bio \*bio\_list, \*\*bio\_tail;*

*/\**

*47. reclaim\_state*

*内存回收*

*\*/*

*struct reclaim\_state \*reclaim\_state;*

*/\**

*48. backing\_dev\_info*

*存放块设备I/O数据流量信息*

*\*/*

*struct backing\_dev\_info \*backing\_dev\_info;*

*/\**

*49. io\_context*

*I/O调度器所使用的信息*

*\*/*

*struct io\_context \*io\_context;*

*/\**

*50. CPUSET功能*

*\*/*

*#ifdef CONFIG\_CPUSETS*

*nodemask\_t mems\_allowed;*

*int cpuset\_mem\_spread\_rotor;*

*#endif*

*/\**

*51. Control Groups*

*\*/*

*#ifdef CONFIG\_CGROUPS*

*struct css\_set \*cgroups;*

*struct list\_head cg\_list;*

*#endif*

*/\**

*52. robust\_list*

*futex同步机制*

*\*/*

*#ifdef CONFIG\_FUTEX*

*struct robust\_list\_head \_\_user \*robust\_list;*

*#ifdef CONFIG\_COMPAT*

*struct compat\_robust\_list\_head \_\_user \*compat\_robust\_list;*

*#endif*

*struct list\_head pi\_state\_list;*

*struct futex\_pi\_state \*pi\_state\_cache;*

*#endif*

*#ifdef CONFIG\_PERF\_EVENTS*

*struct perf\_event\_context \*perf\_event\_ctxp;*

*struct mutex perf\_event\_mutex;*

*struct list\_head perf\_event\_list;*

*#endif*

*/\**

*53. 非一致内存访问(NUMA Non-Uniform Memory Access)*

*\*/*

*#ifdef CONFIG\_NUMA*

*struct mempolicy \*mempolicy; /\* Protected by alloc\_lock \*/*

*short il\_next;*

*#endif*

*/\**

*54. fs\_excl*

*文件系统互斥资源*

*\*/*

*atomic\_t fs\_excl;*

*/\**

*55. rcu*

*RCU链表*

*\*/*

*struct rcu\_head rcu;*

*/\**

*56. splice\_pipe*

*管道*

*\*/*

*struct pipe\_inode\_info \*splice\_pipe;*

*/\**

*57. delays*

*延迟计数*

*\*/*

*#ifdef CONFIG\_TASK\_DELAY\_ACCT*

*struct task\_delay\_info \*delays;*

*#endif*

*/\**

*58. make\_it\_fail*

*fault injection*

*\*/*

*#ifdef CONFIG\_FAULT\_INJECTION*

*int make\_it\_fail;*

*#endif*

*/\**

*59. dirties*

*FLoating proportions*

*\*/*

*struct prop\_local\_single dirties;*

*/\**

*60. Infrastructure for displayinglatency*

*\*/*

*#ifdef CONFIG\_LATENCYTOP*

*int latency\_record\_count;*

*struct latency\_record latency\_record[LT\_SAVECOUNT];*

*#endif*

*/\**

*61. time slack values，常用于poll和select函数*

*\*/*

*unsigned long timer\_slack\_ns;*

*unsigned long default\_timer\_slack\_ns;*

*/\**

*62. scm\_work\_list*

*socket控制消息(control message)*

*\*/*

*struct list\_head \*scm\_work\_list;*

*/\**

*63. ftrace跟踪器*

*\*/*

*#ifdef CONFIG\_FUNCTION\_GRAPH\_TRACER*

*int curr\_ret\_stack;*

*struct ftrace\_ret\_stack \*ret\_stack;*

*unsigned long long ftrace\_timestamp;*

*atomic\_t trace\_overrun;*

*atomic\_t tracing\_graph\_pause;*

*#endif*

*#ifdef CONFIG\_TRACING*

*unsigned long trace;*

*unsigned long trace\_recursion;*

*#endif*

*};*

## 2.2 进程调度

正文

## 3.1 关于进程的一些思考题？

### 3.3.3 XXXXXXXXXXXXXXXXXXXXXXXXXXX

# 第三章内存管理

## 3.1页面机制

一级页表

### 3.3.1一级页表

#### 3.3.1.1 32位一级页表

32位一级页表的缺陷：

页面4k=2^12(页内偏移至少需要12位)

对于32位机器如下理解：4G的内存空间(虚拟内存空间)--页表个数=(2^20)-4G(2^32)/4K(2^12)

需要存储2^20个页表项，一个页表项占用32位（4字节）总共需要内存：2^20\*4(4M)内存

也就是一个进程占用4M的大小，如果多个进程引起更大的空间占用浪费。

|  |
| --- |
| 0 |
| 1 |
| 2 |
| ... |
| 2^20-1 |

一级页表：

|  |  |  |
| --- | --- | --- |
| 序号 | 31~12位 | 11~0位 |
| 0 |  |  |
| 1 |  |  |
| 2 |  |  |
| .... |  |  |
| 2^20-1 |  |  |

#### 3.3.1.2 64位一级页表

### 3.3.2二级页表

#### 3.3.2.1 32位系统二级页表

|  |
| --- |
| 0 |
| 1 |
| 2 |
| 3 |
| ... |
| ... |
| 2^20-1 |

二级页表：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 10位(dir) | 10位(pag) | 12位（offset） |
| 序号 | 31~22 | 21~12 | 11~0 |
| 0 | 0 | 1 | 12 |
| 1 |  |  |  |
| 2 |  |  |  |
| .... |  |  |  |
| .... |  |  |  |
| 1024 |  |  |  |

二级页表原理：将一级页表的20位拆分两部分：10（第一层）+10（第二层）

一个进程需要的最小内存页表是4K=2^10 + 2^10+ 2^12，

在PCB中可以获取一级页目录的开始位置，系统初始化建立一个对应关系。

这样的页目录需要连续的4M空间，

将页目录拆分如下图：

|  |  |
| --- | --- |
| 0#页表 | |
| 0 | … |
| 1 | 762 |
| 1023 | … |

|  |  |
| --- | --- |
| 存放数据 | 物理块号 |
|  | 1 |
|  | 2 |
| 0#页表位置 | 3 |
|  | … |
|  | 762 |
|  | ... |
|  | 2^20-1 |

|  |  |
| --- | --- |
| 页表号 | 内存块号 |
| 0 | 3 |
| 1 | … |
| … | … |
| 1023 | ….. |

|  |  |
| --- | --- |
| 1#页表 | |
| 0 | …. |
| ... |  |
|  |  |
| 1023 | … |

|  |  |
| --- | --- |
| 页号 | 块号 |
| 0 | 2 |
| 1 | 4 |
| ... | ... |
| 1023 | 762 |
| ..... | ... |
| 2^20-1 | .... |

|  |  |
| --- | --- |
| 2#页表 | |
| 0 | … |
| ... |  |
| 1023 | … |



比如：32位系统 一个0000000000 0000000001 000000001111 逻辑地址转换物理地址？

762\*1024+(1111)偏移

#### 3.3.2.2 64位系统多级页表

64位系统需要页表偏移是2^64 = 2^34G = 2^24T 但是虚拟并不是2^34内存空间（如果这么大有点吓人）。注意一个概念long( 32位4字节 && 64位8字节 )

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [63:48] | [47:39] 9 | [38:30] 9 | [29:21] 9 | [20:12] 9 | [11:0]12 |
| 预留 | PGD | PUD | PMD | PTE | Offset |

如果：[63]=1 内核空间地址 && [63]=0 用户空间地址

在linux划分中各占用9位，各有2^9=512个表项



#define PAGE\_SHIFT 12 //也就是偏移量 2^12=4K(一个页面的大小)

问题：1、如何求出PGD基地址？

2、00 01 01 01 01 02 ？物理地址是多少？

00首位”0” 用户空间地址

01(PGD=000000001)索引1

FF 01 01 01 01 02 ？物理地址是多少？

## 3.3 虚拟地址转换物理地址

物理页面的切换就是取出mm中的PGD参数放置在cr3寄存器中。

pgd\_offset() 求出的页目录项低12位没有用保存的是页目录项的一些属性。



[root@localhost ~]# dmesg -c

[ 289.469352] \*\*\*\*\*\*\*start\*\*\*\*\*\*\*\*

[ 289.469356] cr0= 80050033 cr3= 7b3e0000

[ 289.469359] PGDIR\_SHIFT = 39

[ 289.469362] PUD\_SHIFT = 30

[ 289.469364] PMD\_SHIFT = 21

[ 289.469366] PAGE\_SHIFT = 12

[ 289.469369] PTRS\_PER\_PGD = 512

[ 289.469371] PTRS\_PER\_PUD = 512

[ 289.469373] PTRS\_PER\_PMD = 512

[ 289.469375] PTRS\_PER\_PTE = 512

[ 289.469378] PAGE\_MASK = fffffffffffff000

[ 289.469382] vaddr = ffff99fdf65df000

[ 289.469385] pgd\_val = 4051067 , pgd\_index = 133

[ 289.469387] pud\_val = 4055067 , pud\_index = 1f7

[ 289.469390] pmd\_val = 7648a063 , pmd\_index = 1b2

[ 289.469392] pte\_val = 80000000765df063 , pte\_index = 1df

[ 289.469395] page\_addr = 80000000765df000 , page\_offset = 0

[ 289.469397] vaddr = ffff99fdf65df000 , paddr = 80000000765df000

下面是内存的真是存在数值:

第一步：寄存器cr3+PGD\*8B = 0x7b3e0000+0x133\*8 = 0x7b3e998

0x7b3e998地址存放的数据0x4051067

000007B3E0980 0000000000000000 0000000000000000 ................

000007B3E0990 0000000000000000 0000000004051067 ........g.......

000007B3E09A0 0000000000000000 0000000000000000 ................

第二步：基址0x4051000+0x1f7\*8 = 0x4051fb8

0x4051fb8地址存放的数据是0x4055067

0000004051FA0 0000000000000000 0000000000000000 ................

0000004051FB0 0000000004052067 0000000004055067 g......gP......

0000004051FC0 0000000000000000 0000000000000000 ................

第三步：0x4055000+0x1b2\*8=0x4055D90

0x4055D90地址存放的数据是0x7648A063

0000004055D80 80000000760000E3 000000007638C063 ...v....c.8v....

0000004055D90 000000007648A063 00000000327E8063 c.Hv....c.~2....

0000004055DA0 80000000768000E3 8000000076A000E3 ...v.......v....

第四步：0x7648A000+0x1df\*8 = 0x7648Aef8

0x7648Aef8地址存放的数据是0x765DF063

000007648AEE0 80000000765DC063 80000000765DD063 c.]v....c.]v....

000007648AEF0 80000000765DE063 80000000765DF063 c.]v....c.]v....

000007648AF00 80000000765E0063 80000000765E1063 c.^v....c.^v....

第五步：0x765DF000+0x0\*0 =0x765DF000

00000765DEFF0 0000000000000000 0000000000000000 ...............

00000765DF000 726F77206C6C6568 206D6F726620646C hellworld from

00000765DF010 00006C656E72656B 0000000000000030 kernel..0.......

真实的物理地址是：0x765DF000

## 3.3 内存管理

### 3.3.1 栈追踪 stack\_trace

struct [**stack\_trace**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/stack_trace) {

unsigned int [**nr\_entries**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/nr_entries), [**max\_entries**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/max_entries);

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) \*[**entries**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/entries);

int [**skip**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/skip); */\* input argument: How many entries to skip\*/*

};

引用地址：<http://kerneltravel.net>

### 3.3.1内存管理 mm\_struct

mm\_struct 被称为内存描述符，对整个进程的空间的描述，但是mm\_struct包含多个vm\_area\_struct（虚拟区域）如下图所示：



bss属于所有未初始化存放的位置



struct mm\_struct {

struct vm\_area\_struct \* mmap; //指向虚拟区间(VMA)的链表

struct rb\_root mm\_rb; //指向线性区对象红黑树的根

struct vm\_area\_struct \* mmap\_cache; //指向最近找到的虚拟区间

unsigned long(\*get\_unmapped\_area) (struct file \*filp,

unsigned long addr, unsigned long len,

unsigned long pgoff, unsigned long flags);//在进程地址空间中搜索有效线性地址区

unsigned long(\*get\_unmapped\_exec\_area) (struct file \*filp,

unsigned long addr, unsigned long len,

unsigned long pgoff, unsigned long flags);

void(\*unmap\_area) (struct mm\_struct \*mm, unsigned long addr);//释放线性地址区间时调用的方法

unsigned long mmap\_base; /\* base of mmap area \*/

unsigned long task\_size; /\* size of task vm space \*/

unsigned long cached\_hole\_size;

unsigned long free\_area\_cache; //内核从这个地址开始搜索进程地址空间中线性地址的空闲区域

pgd\_t \* pgd; //指向页全局目录

atomic\_t mm\_users; //次使用计数器，使用这块空间的个数

atomic\_t mm\_count; //主使用计数器

int map\_count; //线性的个数

struct rw\_semaphore mmap\_sem; //线性区的读/写信号量

spinlock\_t page\_table\_lock; //线性区的自旋锁和页表的自旋锁

struct list\_head mmlist; //指向内存描述符链表中的相邻元素

/\* Special counters, in some configurations protected by the

\* page\_table\_lock, in other configurations by being atomic.

\*/

mm\_counter\_t \_file\_rss; //mm\_counter\_t代表的类型实际是typedef atomic\_long\_t

mm\_counter\_t \_anon\_rss;

mm\_counter\_t \_swap\_usage;

unsigned long hiwater\_rss; //进程所拥有的最大页框数

unsigned long hiwater\_vm; //进程线性区中最大页数

unsigned long total\_vm, locked\_vm, shared\_vm, exec\_vm;

//total\_vm 进程地址空间的大小(页数）

//locked\_vm 锁住而不能换出的页的个数

//shared\_vm 共享文件内存映射中的页数

unsigned long stack\_vm, reserved\_vm, def\_flags, nr\_ptes;

//stack\_vm 用户堆栈中的页数

//reserved\_vm 在保留区中的页数或者在特殊线性区中的页数

//def\_flags 线性区默认的访问标志

//nr\_ptes 进程的页表数

unsigned long start\_code, end\_code, start\_data, end\_data;

//start\_code 可执行代码的起始地址

//end\_code 可执行代码的最后地址

//start\_data已初始化数据的起始地址

// end\_data已初始化数据的最后地址

unsigned long start\_brk, brk, start\_stack;

//start\_stack堆的起始位置

//brk堆的当前的最后地址

//用户堆栈的起始地址

unsigned long arg\_start, arg\_end, env\_start, env\_end;

//arg\_start 命令行参数的起始地址

//arg\_end命令行参数的起始地址

//env\_start环境变量的起始地址

//env\_end环境变量的最后地址

unsigned long saved\_auxv[AT\_VECTOR\_SIZE]; /\* for /proc/PID/auxv \*/

struct linux\_binfmt \*binfmt;

cpumask\_t cpu\_vm\_mask; //用于惰性TLB交换的位掩码

/\* Architecture-specific MM context \*/

mm\_context\_t context; //指向有关特定结构体系信息的表

unsigned int faultstamp;

unsigned int token\_priority;

unsigned int last\_interval;

unsigned long flags; /\* Must use atomic bitops to access the bits \*/

struct core\_state \*core\_state; /\* coredumping support \*/

#ifdef CONFIG\_AIO

spinlock\_t ioctx\_lock; //用于保护异步I/O上下文链表的锁

struct hlist\_head ioctx\_list;//异步I/O上下文

#endif

#ifdef CONFIG\_MM\_OWNER

struct task\_struct \*owner;

#endif

#ifdef CONFIG\_PROC\_FS

unsigned long num\_exe\_file\_vmas;

#endif

#ifdef CONFIG\_MMU\_NOTIFIER

struct mmu\_notifier\_mm \*mmu\_notifier\_mm;

#endif

#ifdef CONFIG\_TRANSPARENT\_HUGEPAGE

pgtable\_t pmd\_huge\_pte; /\* protected by page\_table\_lock \*/

#endif

#ifdef \_\_GENKSYMS\_\_

unsigned long rh\_reserved[2];

#else

//有多少任务分享这个mm OOM\_DISABLE

union {

unsigned long rh\_reserved\_aux;

atomic\_t oom\_disable\_count;

};

/\* base of lib map area (ASCII armour) \*/

unsigned long shlib\_base;

#endif

};

### 3.3.3内存管理 vm\_area\_struct

是最小的内存管理结构，表示一段属性一样的内存区域。

struct vm\_area\_struct {

/\* The first cache line has the info for VMA tree walking.

第一个缓存行具有VMA树移动的信息\*/

unsigned long vm\_start; /\* vm\_mm内的起始地址。 \*/  
unsigned long vm\_end; /\* 在vm\_mm内结束地址之后的第一个字节的地址。 \*

/\* linked list of VM areas per task, sorted by address

每个任务的VM区域的链接列表，按地址排序\*/

struct vm\_area\_struct \*vm\_next, \*vm\_prev;

struct rb\_node vm\_rb;

/\*

此VMA左侧最大的可用内存间隙（以字节为单位）。

在此VMA和vma-> vm\_prev之间，

或者在VMA rbtree中我们下面的一个VMA与其->vm\_prev之间。

这有助于get\_unmapped\_area找到合适大小的空闲区域。

\*/

unsigned long rb\_subtree\_gap;

/\* Second cache line starts here.

第二个缓存行从这里开始\*/

struct mm\_struct \*vm\_mm; /\* 我们所属的address space\*/

pgprot\_t vm\_page\_prot; /\* 此VMA的访问权限 \*/

unsigned long vm\_flags; /\* Flags, see mm.h. \*/

/\*

对于具有地址空间（address apace）和后备存储(backing store)的区域，

链接到address\_space->i\_mmap间隔树，或者链接到address\_space-> i\_mmap\_nonlinear列表中的vma。

\*/

union {

struct {

struct rb\_node rb;

unsigned long rb\_subtree\_last;

} linear;

struct list\_head nonlinear;

} shared;

/\*

在其中一个文件页面的COW之后，文件的MAP\_PRIVATE vma可以在i\_mmap树和anon\_vma列表中。

MAP\_SHARED vma只能位于i\_mmap树中。

匿名MAP\_PRIVATE，堆栈或brk vma（带有NULL文件）只能位于anon\_vma列表中。

\*/

struct list\_head anon\_vma\_chain; /\* Serialized by mmap\_sem & \* page\_table\_lock

由mmap\_sem和\* page\_table\_lock序列化\*/

struct anon\_vma \*anon\_vma; /\* Serialized by page\_table\_lock 由page\_table\_lock序列化\*/

/\* 用于处理此结构体的函数指针 \*/

const struct vm\_operations\_struct \*vm\_ops;

/\* 后备存储（backing store）的信息: \*/

unsigned long vm\_pgoff; /\* 以PAGE\_SIZE为单位的偏移量（在vm\_file中），\*不是\* PAGE\_CACHE\_SIZE\*/

struct file \* vm\_file; /\* 我们映射到文件（可以为NULL）\*/

void \* vm\_private\_data; /\* 是vm\_pte（共享内存） \*/

#ifndef CONFIG\_MMU

struct vm\_region \*vm\_region; /\* NOMMU映射区域 \*/

#endif

#ifdef CONFIG\_NUMA

struct mempolicy \*vm\_policy; /\* 针对VMA的NUMA政策 \*/

#endif

#### 3.3.3.1 实验代码

打印一个进程所有的vma区域大小-和smaps内容想对应

static void printit(struct task\_struct \*tsk)

{

struct mm\_struct \*mm;

struct vm\_area\_struct \*vma;

int j = 0;

unsigned long start, end, length;

mm = tsk->mm;

pr\_info("mm\_struct addr = 0x%p\n", mm);//

vma = mm->mmap;

/\* protect from simultaneous modification \*/

down\_read(&mm->mmap\_sem);//锁信号

pr\_info("vmas: vma start end length\n");

//vma虚拟地址的首地址（虚拟地址链表的首地址）vma = mm->mmap;

while (vma) {

j++;

start = vma->vm\_start;

end = vma->vm\_end;

length = end - start;

pr\_info("%6d: %16p %12lx %12lx %8ld\n",

j, vma, start, end, length);

vma = vma->vm\_next;//指向下个vma的起始地址

}

up\_read(&mm->mmap\_sem);

}

### 3.3.3内存管理 page struct

<http://blog.chinaunix.net/uid-30282771-id-5176971.html>

https://wenku.baidu.com/view/9d3ba60703d8ce2f006623b2.html

/\*

\* Each physical page in the system has a struct page associated with

\* it to keep track of whatever it is we are using the page for at the

\* moment. Note that we have no way to track which tasks are using

\* a page, though if it is a pagecache page, rmap structures can tell us

\* who is mapping it.

\*

\* The objects in struct page are organized in double word blocks in

\* order to allows us to use atomic double word operations on portions

\* of struct page. That is currently only used by slub but the arrangement

\* allows the use of atomic double word operations on the flags/mapping

\* and lru list pointers also.

\*/

struct page {

/\* First double word block \*/

unsigned long flags; /\* Atomic flags, some possibly updated asynchronously

描述page的状态和其他信息 \*/

union

{

struct address\_space \*mapping; /\* If low bit clear, points to

\* inode address\_space, or NULL.

\* If page mapped as anonymous

\* memory, low bit is set, and

\* it points to anon\_vma object:

\* see PAGE\_MAPPING\_ANON below.

\*/

void \*s\_mem; /\* slab first object \*/

atomic\_t compound\_mapcount; /\* first tail page \*/

/\* page\_deferred\_list().next -- second tail page \*/

};

/\* Second double word \*/

struct {

union {

pgoff\_t index; /\* Our offset within mapping.

在映射的虚拟空间（vma\_area）内的偏移；

一个文件可能只映射一部分，假设映射了1M的空间，

index指的是在1M空间内的偏移，而不是在整个文件内的偏移。 \*/

void \*freelist; /\* sl[aou]b first free object \*/

/\* page\_deferred\_list().prev -- second tail page \*/

};

union {

#if defined(CONFIG\_HAVE\_CMPXCHG\_DOUBLE) && \

defined(CONFIG\_HAVE\_ALIGNED\_STRUCT\_PAGE)

/\* Used for cmpxchg\_double in slub \*/

unsigned long counters;

#else

/\*

\* Keep \_refcount separate from slub cmpxchg\_double

\* data. As the rest of the double word is protected by

\* slab\_lock but \_refcount is not.

\*/

unsigned counters;

#endif

struct {

union {

/\*

\* Count of ptes mapped in mms, to show

\* when page is mapped & limit reverse

\* map searches.

\* 页映射计数器

\*/

atomic\_t \_mapcount;

struct { /\* SLUB \*/

unsigned inuse:16;

unsigned objects:15;

unsigned frozen:1;

};

int units; /\* SLOB \*/

};

/\*

\* Usage count, \*USE WRAPPER FUNCTION\*

\* when manual accounting. See page\_ref.h

\* 页引用计数器

\*/

atomic\_t \_refcount;

};

unsigned int active; /\* SLAB \*/

};

};

/\*

\* Third double word block

\*

\* WARNING: bit 0 of the first word encode PageTail(). That means

\* the rest users of the storage space MUST NOT use the bit to

\* avoid collision and false-positive PageTail().

\*/

union {

struct list\_head lru; /\* Pageout list, eg. active\_list

\* protected by zone->lru\_lock !

\* Can be used as a generic list

\* by the page owner.

\*/

struct dev\_pagemap \*pgmap; /\* ZONE\_DEVICE pages are never on an

\* lru or handled by a slab

\* allocator, this points to the

\* hosting device page map.

\*/

struct { /\* slub per cpu partial pages \*/

struct page \*next; /\* Next partial slab \*/

#ifdef CONFIG\_64BIT

int pages; /\* Nr of partial slabs left \*/

int pobjects; /\* Approximate # of objects \*/

#else

short int pages;

short int pobjects;

#endif

};

struct rcu\_head rcu\_head; /\* Used by SLAB

\* when destroying via RCU

\*/

/\* Tail pages of compound page \*/

struct {

unsigned long compound\_head; /\* If bit zero is set \*/

/\* First tail page only \*/

#ifdef CONFIG\_64BIT

/\*

\* On 64 bit system we have enough space in struct page

\* to encode compound\_dtor and compound\_order with

\* unsigned int. It can help compiler generate better or

\* smaller code on some archtectures.

\*/

unsigned int compound\_dtor;

unsigned int compound\_order;

#else

unsigned short int compound\_dtor;

unsigned short int compound\_order;

#endif

};

#if defined(CONFIG\_TRANSPARENT\_HUGEPAGE) && USE\_SPLIT\_PMD\_PTLOCKS

struct {

unsigned long \_\_pad; /\* do not overlay pmd\_huge\_pte

\* with compound\_head to avoid

\* possible bit 0 collision.

\*/

pgtable\_t pmd\_huge\_pte; /\* protected by page->ptl \*/

};

#endif

};

/\* Remainder is not double word aligned \*/

union {

unsigned long private; /\* Mapping-private opaque data:

\* usually used for buffer\_heads

\* if PagePrivate set; used for

\* swp\_entry\_t if PageSwapCache;

\* indicates order in the buddy

\* system if PG\_buddy is set.

\* 私有数据指针，由应用场景确定其具体的含义

\*/

#if USE\_SPLIT\_PTE\_PTLOCKS

#if ALLOC\_SPLIT\_PTLOCKS

spinlock\_t \*ptl;

#else

spinlock\_t ptl;

#endif

#endif

struct kmem\_cache \*slab\_cache; /\* SL[AU]B: Pointer to slab \*/

};

#ifdef CONFIG\_MEMCG

struct mem\_cgroup \*mem\_cgroup;

#endif

/\*

\* On machines where all RAM is mapped into kernel address space,

\* we can simply calculate the virtual address. On machines with

\* highmem some memory is mapped into kernel virtual memory

\* dynamically, so we need a place to store that address.

\* Note that this field could be 16 bits on x86 ... ;)

\*

\* Architectures with slow multiplication can define

\* WANT\_PAGE\_VIRTUAL in asm/page.h

\*/

#if defined(WANT\_PAGE\_VIRTUAL)

void \*virtual; /\* Kernel virtual address (NULL if

not kmapped, ie. highmem) \*/

#endif /\* WANT\_PAGE\_VIRTUAL \*/

#ifdef CONFIG\_KMEMCHECK

/\*

\* kmemcheck wants to track the status of each byte in a page; this

\* is a pointer to such a status block. NULL if not tracked.

\*/

void \*shadow;

#endif

#ifdef LAST\_CPUPID\_NOT\_IN\_PAGE\_FLAGS

int \_last\_cpupid;

#endif

}

/\*

\* The struct page can be forced to be double word aligned so that atomic ops

\* on double words work. The SLUB allocator can make use of such a feature.

\*/

#ifdef CONFIG\_HAVE\_ALIGNED\_STRUCT\_PAGE

\_\_aligned(2 \* sizeof(unsigned long))

#endif

因为内核会为每一个物理页帧创建一个struct page的结构体，因此要保证page结构体足够的小，否则仅struct page就要占用大量的内存。  
    出于节省内存的考虑，struct page中使用了大量的联合体union。下面仅对常用的一些的字段做说明：  
    1) **flags**：描述page的状态和其他信息，如图：  
      
     主要分为4部分，其中标志位flag向高位增长，其余位字段向低位增长，中间存在空闲位。  
     section：主要用于稀疏内存模型SPARSEMEM，可忽略。  
     node：NUMA节点号，标识该page属于哪一个节点。  
     zone：内存域标志，标识该page属于哪一个zone。  
     flag：page的状态标识，常用的有：  
        PG\_locked：page被锁定，说明有使用者正在操作该page。  
        PG\_error：状态标志，表示涉及该page的IO操作发生了错误。  
        PG\_referenced：表示page刚刚被访问过。  
        PG\_active：page处于inactive LRU链表。PG\_active和PG\_referenced一起控制该page的活跃程度，这在内存回收时将会非常有用。  
        PG\_uptodate：表示page的数据已经与后备存储器是同步的，是最新的。  
        PG\_dirty：与后备存储器中的数据相比，该page的内容已经被修改。  
        PG\_lru：表示该page处于LRU链表上。  
        PG\_slab：该page属于slab分配器。  
        PG\_reserved：设置该标志，防止该page被交换到swap。  
        PG\_private：如果page中的private成员非空，则需要设置该标志。参考6)对private的解释。  
        PG\_writeback：page中的数据正在被回写到后备存储器。  
        PG\_swapcache：表示该page处于swap cache中。  
        PG\_mappedtodisk：表示page中的数据在后备存储器中有对应。  
        PG\_reclaim：表示该page要被回收。当PFRA决定要回收某个page后，需要设置该标志。  
        PG\_swapbacked：该page的后备存储器是swap。  
        PG\_unevictable：该page被锁住，不能交换，并会出现在LRU\_UNEVICTABLE链表中，它包括的几种page：ramdisk或ramfs使用的页、  
            shm\_locked、mlock锁定的页。  
        PG\_mlocked：该page在vma中被锁定，一般是通过系统调用mlock()锁定了一段内存。  
  
    内核中提供了一些标准宏，用来检查、操作某些特定的比特位，如：  
        -> PageXXX(page)：检查page是否设置了PG\_XXX位  
        -> SetPageXXX(page)：设置page的PG\_XXX位  
        -> ClearPageXXX(page)：清除page的PG\_XXX位  
        -> TestSetPageXXX(page)：设置page的PG\_XXX位，并返回原值  
        -> TestClearPageXXX(page)：清除page的PG\_XXX位，并返回原值  
  
    2)**\_count：**引用计数，表示内核中引用该page的次数，如果要操作该page，引用计数会+1，操作完成-1。当该值为0时，表示没有引用该page的位置，所以该page可以被解除映射，这往往在内存回收时是有用的。  
  
    3) **\_mapcount：**被页表映射的次数，也就是说该page同时被多少个进程共享。初始值为-1，如果只被一个进程的页表映射了，该值为0 。如果该page处于伙伴系统中，该值为PAGE\_BUDDY\_MAPCOUNT\_VALUE（-128），内核通过判断该值是否为PAGE\_BUDDY\_MAPCOUNT\_VALUE来确定该page是否属于伙伴系统。  
    注意区分\_count和\_mapcount，\_mapcount表示的是映射次数，而\_count表示的是使用次数；被映射了不一定在使用，但要使用必须先映射。  
  
    4) **mapping：**有三种含义  
        a: 如果mapping = 0，说明该page属于交换缓存（swap cache）；当需要使用地址空间时会指定交换分区的地址空间swapper\_space。  
        b: 如果mapping != 0，bit[0] = 0，说明该page属于页缓存或文件映射，mapping指向文件的地址空间address\_space。  
        c: 如果mapping != 0，bit[0] != 0，说明该page为匿名映射，mapping指向struct anon\_vma对象。  
        通过mapping恢复anon\_vma的方法：anon\_vma = (struct anon\_vma \*)(mapping - PAGE\_MAPPING\_ANON)。  
  
    5)**index：**在映射的虚拟空间（vma\_area）内的偏移；一个文件可能只映射一部分，假设映射了1M的空间，index指的是在1M空间内的偏移，而不是在整个文件内的偏移。  
  
    6) **private：**私有数据指针，由应用场景确定其具体的含义：  
        a：如果设置了PG\_private标志，表示buffer\_heads；  
        b：如果设置了PG\_swapcache标志，private存储了该page在交换分区中对应的位置信息swp\_entry\_t。  
        c：如果\_mapcount = PAGE\_BUDDY\_MAPCOUNT\_VALUE，说明该page位于伙伴系统，private存储该伙伴的阶。  
  
    7) **lru：**链表头，主要有3个用途：  
        a：page处于伙伴系统中时，用于链接相同阶的伙伴（只使用伙伴中的第一个page的lru即可达到目的）。  
        b：page属于slab时，page->lru.next指向page驻留的的缓存的管理结构，page->lru.prec指向保存该page的slab的管理结构。  
        c：page被用户态使用或被当做页缓存使用时，用于将该page连入zone中相应的lru链表，供内存回收时使用。

### 3.3.3 内存管理 vm\_operations\_struct

struct [**vm\_operations\_struct**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_operations_struct) {

void (\*[**open**](https://elixir.bootlin.com/linux/v5.0.21/ident/open))(struct [**vm\_area\_struct**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_area_struct) \* [**area**](https://elixir.bootlin.com/linux/v5.0.21/ident/area));

void (\*[**close**](https://elixir.bootlin.com/linux/v5.0.21/ident/close))(struct [**vm\_area\_struct**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_area_struct) \* [**area**](https://elixir.bootlin.com/linux/v5.0.21/ident/area));

int (\*[**split**](https://elixir.bootlin.com/linux/v5.0.21/ident/split))(struct [**vm\_area\_struct**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_area_struct) \* [**area**](https://elixir.bootlin.com/linux/v5.0.21/ident/area), unsigned long addr);

int (\*[**mremap**](https://elixir.bootlin.com/linux/v5.0.21/ident/mremap))(struct [**vm\_area\_struct**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_area_struct) \* [**area**](https://elixir.bootlin.com/linux/v5.0.21/ident/area));

[**vm\_fault\_t**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_fault_t) (\*[**fault**](https://elixir.bootlin.com/linux/v5.0.21/ident/fault))(struct [**vm\_fault**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_fault) \*vmf);

[**vm\_fault\_t**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_fault_t) (\*huge\_fault)(struct [**vm\_fault**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_fault) \*vmf,

enum [**page\_entry\_size**](https://elixir.bootlin.com/linux/v5.0.21/ident/page_entry_size) pe\_size);

void (\*[**map\_pages**](https://elixir.bootlin.com/linux/v5.0.21/ident/map_pages))(struct [**vm\_fault**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_fault) \*vmf,

[**pgoff\_t**](https://elixir.bootlin.com/linux/v5.0.21/ident/pgoff_t) start\_pgoff, [**pgoff\_t**](https://elixir.bootlin.com/linux/v5.0.21/ident/pgoff_t) end\_pgoff);

unsigned long (\*[**pagesize**](https://elixir.bootlin.com/linux/v5.0.21/ident/pagesize))(struct [**vm\_area\_struct**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_area_struct) \* [**area**](https://elixir.bootlin.com/linux/v5.0.21/ident/area));

*/\* notification that a previously read-only page is about to become*

*\* writable, if an error is returned it will cause a SIGBUS \*/*

[**vm\_fault\_t**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_fault_t) (\*page\_mkwrite)(struct [**vm\_fault**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_fault) \*vmf);

*/\* same as page\_mkwrite when using VM\_PFNMAP|VM\_MIXEDMAP \*/*

[**vm\_fault\_t**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_fault_t) (\*pfn\_mkwrite)(struct [**vm\_fault**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_fault) \*vmf);

*/\* called by access\_process\_vm when get\_user\_pages() fails, typically*

*\* for use by special VMAs that can switch between memory and hardware*

*\*/*

int (\*[**access**](https://elixir.bootlin.com/linux/v5.0.21/ident/access))(struct [**vm\_area\_struct**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_area_struct) \*vma, unsigned long addr,

void \*buf, int len, int [**write**](https://elixir.bootlin.com/linux/v5.0.21/ident/write));

*/\* Called by the /proc/PID/maps code to ask the vma whether it*

*\* has a special name. Returning non-NULL will also cause this*

*\* vma to be dumped unconditionally. \*/*

const char \*(\*name)(struct [**vm\_area\_struct**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_area_struct) \*vma);

#[**ifdef**](https://elixir.bootlin.com/linux/v5.0.21/ident/ifdef) CONFIG\_NUMA

*/\**

*\* set\_policy() op must add a reference to any non-NULL @new mempolicy*

*\* to hold the policy upon return. Caller should pass NULL @new to*

*\* remove a policy and fall back to surrounding context--i.e. do not*

*\* install a MPOL\_DEFAULT policy, nor the task or system default*

*\* mempolicy.*

*\*/*

int (\*set\_policy)(struct [**vm\_area\_struct**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_area_struct) \*vma, struct [**mempolicy**](https://elixir.bootlin.com/linux/v5.0.21/ident/mempolicy) \*[**new**](https://elixir.bootlin.com/linux/v5.0.21/ident/new));

*/\**

*\* get\_policy() op must add reference [mpol\_get()] to any policy at*

*\* (vma,addr) marked as MPOL\_SHARED. The shared policy infrastructure*

*\* in mm/mempolicy.c will do this automatically.*

*\* get\_policy() must NOT add a ref if the policy at (vma,addr) is not*

*\* marked as MPOL\_SHARED. vma policies are protected by the mmap\_sem.*

*\* If no [shared/vma] mempolicy exists at the addr, get\_policy() op*

*\* must return NULL--i.e., do not "fallback" to task or system default*

*\* policy.*

*\*/*

struct [**mempolicy**](https://elixir.bootlin.com/linux/v5.0.21/ident/mempolicy) \*(\*[**get\_policy**](https://elixir.bootlin.com/linux/v5.0.21/ident/get_policy))(struct [**vm\_area\_struct**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_area_struct) \*vma,

unsigned long addr);

#endif

*/\**

*\* Called by vm\_normal\_page() for special PTEs to find the*

*\* page for @addr. This is useful if the default behavior*

*\* (using pte\_page()) would not find the correct page.*

*\*/*

struct page \*(\*find\_special\_page)(struct [**vm\_area\_struct**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_area_struct) \*vma,

unsigned long addr);

};

### 3.3.3 内存管理 vm\_fault

存放与缺页相关的参数

Address 产生缺页的线性地址

*/\**

*\* vm\_fault is filled by the the pagefault handler and passed to the vma's*

*\* ->fault function. The vma's ->fault is responsible for returning a bitmask*

*\* of VM\_FAULT\_xxx flags that give details about how the fault was handled.*

*\**

*\* MM layer fills up gfp\_mask for page allocations but fault handler might*

*\* alter it if its implementation requires a different allocation context.*

*\**

*\* pgoff should be used in favour of virtual\_address, if possible.*

*\*/*

struct [**vm\_fault**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_fault) {

struct [**vm\_area\_struct**](https://elixir.bootlin.com/linux/v5.0.21/ident/vm_area_struct) \*vma; */\* Target VMA \*/*

unsigned int flags; */\* FAULT\_FLAG\_xxx flags \*/*

[**gfp\_t**](https://elixir.bootlin.com/linux/v5.0.21/ident/gfp_t) gfp\_mask; */\* gfp mask to be used for allocations \*/*

[**pgoff\_t**](https://elixir.bootlin.com/linux/v5.0.21/ident/pgoff_t) pgoff; */\* Logical page offset based on vma \*/*

address 产生缺页的线性地址

（pmd pud pte(页表项)）

unsigned long [**address**](https://elixir.bootlin.com/linux/v5.0.21/ident/address); */\* Faulting virtual address \*/（产生缺页的线性地址）*

[**pmd\_t**](https://elixir.bootlin.com/linux/v5.0.21/ident/pmd_t) \*[**pmd**](https://elixir.bootlin.com/linux/v5.0.21/ident/pmd); */\* Pointer to pmd entry matching*

*\* the 'address' \*/*

[**pud\_t**](https://elixir.bootlin.com/linux/v5.0.21/ident/pud_t) \*pud; */\* Pointer to pud entry matching*

*\* the 'address'*

*\*/*

[**pte\_t**](https://elixir.bootlin.com/linux/v5.0.21/ident/pte_t) orig\_pte; */\* Value of PTE at the time of fault \*/*

struct page \*cow\_page; */\* Page handler may use for COW fault \*/*

struct [**mem\_cgroup**](https://elixir.bootlin.com/linux/v5.0.21/ident/mem_cgroup) \*memcg; */\* Cgroup cow\_page belongs to \*/*

struct page \*page; */\* ->fault handlers should return a*

*\* page here, unless VM\_FAULT\_NOPAGE*

*\* is set (which is also implied by*

*\* VM\_FAULT\_ERROR).*

*\*/*

*/\* These three entries are valid only while holding ptl lock \*/*

[**pte\_t**](https://elixir.bootlin.com/linux/v5.0.21/ident/pte_t) \*[**pte**](https://elixir.bootlin.com/linux/v5.0.21/ident/pte); */\* Pointer to pte entry matching*

*\* the 'address'. NULL if the page*

*\* table hasn't been allocated.*

*\*/*

[**spinlock\_t**](https://elixir.bootlin.com/linux/v5.0.21/ident/spinlock_t) \*ptl; */\* Page table lock.*

*\* Protects pte page table if 'pte'*

*\* is not NULL, otherwise pmd.*

*\*/*

[**pgtable\_t**](https://elixir.bootlin.com/linux/v5.0.21/ident/pgtable_t) prealloc\_pte; */\* Pre-allocated pte page table.*

*\* vm\_ops->map\_pages() calls*

*\* alloc\_set\_pte() from atomic context.*

*\* do\_fault\_around() pre-allocates*

*\* page table to avoid allocation from*

*\* atomic context.*

*\*/*

};

### 3.3.3 内存管理 file\_operations

每个字符设备文件，都需要一个设备文件操作字符集（file\_operations）.

struct [file\_operations](http://lxr.linux.no/linux+*/+code=file_operations) {

struct [module](http://lxr.linux.no/linux+*/+code=module) \*[owner](http://lxr.linux.no/linux+*/+code=owner);

[loff\_t](http://lxr.linux.no/linux+*/+code=loff_t) (\*[llseek](http://lxr.linux.no/linux+*/+code=llseek)) (struct [file](http://lxr.linux.no/linux+*/+code=file) \*, [loff\_t](http://lxr.linux.no/linux+*/+code=loff_t), int);

[ssize\_t](http://lxr.linux.no/linux+*/+code=ssize_t) (\*[read](http://lxr.linux.no/linux+*/+code=read)) (struct [file](http://lxr.linux.no/linux+*/+code=file) \*, char [\_\_user](http://lxr.linux.no/linux+*/+code=__user) \*, [size\_t](http://lxr.linux.no/linux+*/+code=size_t), [loff\_t](http://lxr.linux.no/linux+*/+code=loff_t) \*);

[ssize\_t](http://lxr.linux.no/linux+*/+code=ssize_t) (\*[write](http://lxr.linux.no/linux+*/+code=write)) (struct [file](http://lxr.linux.no/linux+*/+code=file) \*, const char [\_\_user](http://lxr.linux.no/linux+*/+code=__user) \*, [size\_t](http://lxr.linux.no/linux+*/+code=size_t), [loff\_t](http://lxr.linux.no/linux+*/+code=loff_t) \*);

[ssize\_t](http://lxr.linux.no/linux+*/+code=ssize_t) (\*[aio\_read](http://lxr.linux.no/linux+*/+code=aio_read)) (struct [kiocb](http://lxr.linux.no/linux+*/+code=kiocb) \*, const struct [iovec](http://lxr.linux.no/linux+*/+code=iovec) \*, unsigned long, [loff\_t](http://lxr.linux.no/linux+*/+code=loff_t));

[ssize\_t](http://lxr.linux.no/linux+*/+code=ssize_t) (\*[aio\_write](http://lxr.linux.no/linux+*/+code=aio_write)) (struct [kiocb](http://lxr.linux.no/linux+*/+code=kiocb) \*, const struct [iovec](http://lxr.linux.no/linux+*/+code=iovec) \*, unsigned long, [loff\_t](http://lxr.linux.no/linux+*/+code=loff_t));

int (\*[readdir](http://lxr.linux.no/linux+*/+code=readdir)) (struct [file](http://lxr.linux.no/linux+*/+code=file) \*, void \*, [filldir\_t](http://lxr.linux.no/linux+*/+code=filldir_t));

unsigned int (\*[poll](http://lxr.linux.no/linux+*/+code=poll)) (struct [file](http://lxr.linux.no/linux+*/+code=file) \*, struct [poll\_table\_struct](http://lxr.linux.no/linux+*/+code=poll_table_struct) \*);

long (\*[unlocked\_ioctl](http://lxr.linux.no/linux+*/+code=unlocked_ioctl)) (struct [file](http://lxr.linux.no/linux+*/+code=file) \*, unsigned int, unsigned long);

long (\*[compat\_ioctl](http://lxr.linux.no/linux+*/+code=compat_ioctl)) (struct [file](http://lxr.linux.no/linux+*/+code=file) \*, unsigned int, unsigned long);

int (\*[mmap](http://lxr.linux.no/linux+*/+code=mmap)) (struct [file](http://lxr.linux.no/linux+*/+code=file) \*, struct [vm\_area\_struct](http://lxr.linux.no/linux+*/+code=vm_area_struct) \*);

int (\*[open](http://lxr.linux.no/linux+*/+code=open)) (struct [inode](http://lxr.linux.no/linux+*/+code=inode) \*, struct [file](http://lxr.linux.no/linux+*/+code=file) \*);

int (\*[flush](http://lxr.linux.no/linux+*/+code=flush)) (struct [file](http://lxr.linux.no/linux+*/+code=file) \*, [fl\_owner\_t](http://lxr.linux.no/linux+*/+code=fl_owner_t) [id](http://lxr.linux.no/linux+*/+code=id));

int (\*[release](http://lxr.linux.no/linux+*/+code=release)) (struct [inode](http://lxr.linux.no/linux+*/+code=inode) \*, struct [file](http://lxr.linux.no/linux+*/+code=file) \*);

int (\*[fsync](http://lxr.linux.no/linux+*/+code=fsync)) (struct [file](http://lxr.linux.no/linux+*/+code=file) \*, [loff\_t](http://lxr.linux.no/linux+*/+code=loff_t), [loff\_t](http://lxr.linux.no/linux+*/+code=loff_t), int [datasync](http://lxr.linux.no/linux+*/+code=datasync));

int (\*[aio\_fsync](http://lxr.linux.no/linux+*/+code=aio_fsync)) (struct [kiocb](http://lxr.linux.no/linux+*/+code=kiocb) \*, int [datasync](http://lxr.linux.no/linux+*/+code=datasync));

int (\*[fasync](http://lxr.linux.no/linux+*/+code=fasync)) (int, struct [file](http://lxr.linux.no/linux+*/+code=file) \*, int);

int (\*[lock](http://lxr.linux.no/linux+*/+code=lock)) (struct [file](http://lxr.linux.no/linux+*/+code=file) \*, int, struct [file\_lock](http://lxr.linux.no/linux+*/+code=file_lock) \*);

[ssize\_t](http://lxr.linux.no/linux+*/+code=ssize_t) (\*[sendpage](http://lxr.linux.no/linux+*/+code=sendpage)) (struct [file](http://lxr.linux.no/linux+*/+code=file) \*, struct [page](http://lxr.linux.no/linux+*/+code=page) \*, int, [size\_t](http://lxr.linux.no/linux+*/+code=size_t), [loff\_t](http://lxr.linux.no/linux+*/+code=loff_t) \*, int);

unsigned long (\*[get\_unmapped\_area](http://lxr.linux.no/linux+*/+code=get_unmapped_area))(struct [file](http://lxr.linux.no/linux+*/+code=file) \*, unsigned long, unsigned long, unsigned long, unsigned long);

int (\*[check\_flags](http://lxr.linux.no/linux+*/+code=check_flags))(int);

int (\*[flock](http://lxr.linux.no/linux+*/+code=flock)) (struct [file](http://lxr.linux.no/linux+*/+code=file) \*, int, struct [file\_lock](http://lxr.linux.no/linux+*/+code=file_lock) \*);

[ssize\_t](http://lxr.linux.no/linux+*/+code=ssize_t) (\*[splice\_write](http://lxr.linux.no/linux+*/+code=splice_write))(struct [pipe\_inode\_info](http://lxr.linux.no/linux+*/+code=pipe_inode_info) \*, struct [file](http://lxr.linux.no/linux+*/+code=file) \*, [loff\_t](http://lxr.linux.no/linux+*/+code=loff_t) \*, [size\_t](http://lxr.linux.no/linux+*/+code=size_t), unsigned int);

[ssize\_t](http://lxr.linux.no/linux+*/+code=ssize_t) (\*[splice\_read](http://lxr.linux.no/linux+*/+code=splice_read))(struct [file](http://lxr.linux.no/linux+*/+code=file) \*, [loff\_t](http://lxr.linux.no/linux+*/+code=loff_t) \*, struct [pipe\_inode\_info](http://lxr.linux.no/linux+*/+code=pipe_inode_info) \*, [size\_t](http://lxr.linux.no/linux+*/+code=size_t), unsigned int);

int (\*[setlease](http://lxr.linux.no/linux+*/+code=setlease))(struct [file](http://lxr.linux.no/linux+*/+code=file) \*, long, struct [file\_lock](http://lxr.linux.no/linux+*/+code=file_lock) \*\*);

long (\*[fallocate](http://lxr.linux.no/linux+*/+code=fallocate))(struct [file](http://lxr.linux.no/linux+*/+code=file) \*[file](http://lxr.linux.no/linux+*/+code=file), int [mode](http://lxr.linux.no/linux+*/+code=mode), [loff\_t](http://lxr.linux.no/linux+*/+code=loff_t) [offset](http://lxr.linux.no/linux+*/+code=offset),[loff\_t](http://lxr.linux.no/linux+*/+code=loff_t) [len](http://lxr.linux.no/linux+*/+code=len));

int (\*[show\_fdinfo](http://lxr.linux.no/linux+*/+code=show_fdinfo))(struct [seq\_file](http://lxr.linux.no/linux+*/+code=seq_file) \*[m](http://lxr.linux.no/linux+*/+code=m), struct [file](http://lxr.linux.no/linux+*/+code=file) \*[f](http://lxr.linux.no/linux+*/+code=f));

};

struct module \*owner

 第一个 file\_operations 成员根本不是一个操作; 它是一个指向拥有这个结构的模块的指针.  
 这个成员用来在它的操作还在被使用时阻止模块被卸载. 几乎所有时间中, 它被简单初始化为   
THIS\_MODULE, 一个在 <linux/module.h> 中定义的宏.这个宏比较复杂，在进行简单学习操作的时候，一般初始化为THIS\_MODULE。

loff\_t (\*llseek) (struct file \* filp , loff\_t  p,  int  orig);  
(指针参数filp为进行读取信息的目标文件结构体指针；参数 p 为文件定位的目标偏移量；参数orig为对文件定位  
的起始地址，这个值可以为文件开头（SEEK\_SET，0,当前位置(SEEK\_CUR,1)，文件末尾(SEEK\_END,2)）  
llseek 方法用作改变文件中的当前读/写位置, 并且新位置作为(正的)返回值.  
loff\_t 参数是一个"long offset", 并且就算在 32位平台上也至少 64 位宽. 错误由一个负返回值指示.  
如果这个函数指针是 NULL, seek 调用会以潜在地无法预知的方式修改 file 结构中的位置计数器( 在"file 结构" 一节中描述).

ssize\_t (\*read) (struct file \* filp, char \_\_user \* buffer, size\_t    size , loff\_t \*  p);  
(指针参数 filp 为进行读取信息的目标文件，指针参数buffer 为对应放置信息的缓冲区（即用户空间内存地址），  
参数size为要读取的信息长度，参数 p 为读的位置相对于文件开头的偏移，在读取信息后，这个指针一般都会移动，移动的值为要读取信息的长度值）  
这个函数用来从设备中获取数据. 在这个位置的一个空指针导致 read 系统调用以 -EINVAL("Invalid argument") 失败.  
 一个非负返回值代表了成功读取的字节数( 返回值是一个 "signed size" 类型, 常常是目标平台本地的整数类型).

ssize\_t (\*aio\_read)(struct kiocb \*  , char \_\_user \*  buffer, size\_t  size ,  loff\_t   p);  
可以看出，这个函数的第一、三个参数和本结构体中的read()函数的第一、三个参数是不同 的，  
异步读写的第三个参数直接传递值，而同步读写的第三个参数传递的是指针，因为AIO从来不需要改变文件的位置。  
异步读写的第一个参数为指向kiocb结构体的指针，而同步读写的第一参数为指向file结构体的指针，每一个I/O请求都对应一个kiocb结构体);  
初始化一个异步读 -- 可能在函数返回前不结束的读操作.如果这个方法是 NULL, 所有的操作会由 read 代替进行(同步地).  
(有关linux异步I/O，可以参考有关的资料，《linux设备驱动开发详解》中给出了详细的解答)

ssize\_t (\*write) (struct file \*  filp, const char \_\_user \*   buffer, size\_t  count, loff\_t \* ppos);  
(参数filp为目标文件结构体指针，buffer为要写入文件的信息缓冲区，count为要写入信息的长度，  
ppos为当前的偏移位置，这个值通常是用来判断写文件是否越界）  
发送数据给设备. 如果 NULL, -EINVAL 返回给调用 write 系统调用的程序. 如果非负, 返回值代表成功写的字节数.  
(注：这个操作和上面的对文件进行读的操作均为阻塞操作）

ssize\_t (\*aio\_write)(struct kiocb \*, const char \_\_user \*  buffer, size\_t  count, loff\_t \* ppos);  
      初始化设备上的一个异步写.参数类型同aio\_read()函数;

int (\*readdir) (struct file \*  filp, void \*, filldir\_t);  
对于设备文件这个成员应当为 NULL; 它用来读取目录, 并且仅对文件系统有用.

unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);  
(这是一个设备驱动中的轮询函数，第一个参数为file结构指针，第二个为轮询表指针）  
这个函数返回设备资源的可获取状态，即POLLIN，POLLOUT，POLLPRI，POLLERR，POLLNVAL等宏的位“或”结果。  
每个宏都表明设备的一种状态，如：POLLIN（定义为0x0001）意味着设备可以无阻塞的读，POLLOUT（定义为0x0004）意味着设备可以无阻塞的写。  
(poll 方法是 3 个系统调用的后端: poll, epoll, 和 select, 都用作查询对一个或多个文件描述符的读或写是否会阻塞.  
 poll 方法应当返回一个位掩码指示是否非阻塞的读或写是可能的, 并且, 可能地, 提供给内核信息用来使调用进程睡眠直到 I/O 变为可能.   
如果一个驱动的 poll 方法为 NULL, 设备假定为不阻塞地可读可写.  
(这里通常将设备看作一个文件进行相关的操作，而轮询操作的取值直接关系到设备的响应情况，可以是阻塞操作结果，同时也可以是非阻塞操作结果）

int (\*ioctl) (struct inode \*inode, struct file \*filp, unsigned int cmd, unsigned long arg);  
(inode 和 filp 指针是对应应用程序传递的文件描述符 fd 的值, 和传递给 open 方法的相同参数.  
cmd 参数从用户那里不改变地传下来, 并且可选的参数 arg 参数以一个 unsigned long 的形式传递, 不管它是否由用户给定为一个整数或一个指针.  
如果调用程序不传递第 3 个参数, 被驱动操作收到的 arg 值是无定义的.  
因为类型检查在这个额外参数上被关闭, 编译器不能警告你如果一个无效的参数被传递给 ioctl, 并且任何关联的错误将难以查找.）  
ioctl 系统调用提供了发出设备特定命令的方法(例如格式化软盘的一个磁道, 这不是读也不是写). 另外, 几个 ioctl 命令被内核识别而不必引用 fops 表.  
 如果设备不提供 ioctl 方法, 对于任何未事先定义的请求(-ENOTTY, "设备无这样的 ioctl"), 系统调用返回一个错误.

int (\*mmap) (struct file \*, struct vm\_area\_struct \*);  
mmap 用来请求将设备内存映射到进程的地址空间. 如果这个方法是 NULL, mmap 系统调用返回 -ENODEV.  
(如果想对这个函数有个彻底的了解，那么请看有关“进程地址空间”介绍的书籍)

int (\*open) (struct inode \* inode , struct file \*  filp ) ;  
(inode 为文件节点,这个节点只有一个，无论用户打开多少个文件，都只是对应着一个inode结构；  
但是filp就不同，只要打开一个文件，就对应着一个file结构体，file结构体通常用来追踪文件在运行时的状态信息）  
 尽管这常常是对设备文件进行的第一个操作, 不要求驱动声明一个对应的方法. 如果这个项是 NULL, 设备打开一直成功, 但是你的驱动不会得到通知.  
与open()函数对应的是release()函数。

int (\*flush) (struct file \*);  
flush 操作在进程关闭它的设备文件描述符的拷贝时调用; 它应当执行(并且等待)设备的任何未完成的操作.  
这个必须不要和用户查询请求的 fsync 操作混淆了. 当前, flush 在很少驱动中使用;  
 SCSI 磁带驱动使用它, 例如, 为确保所有写的数据在设备关闭前写到磁带上. 如果 flush 为 NULL, 内核简单地忽略用户应用程序的请求.

int (\*release) (struct inode \*, struct file \*);  
release ()函数当最后一个打开设备的用户进程执行close()系统调用的时候，内核将调用驱动程序release()函数：  
void release(struct inode inode,struct file \*file),release函数的主要任务是清理未结束的输入输出操作，释放资源，用户自定义排他标志的复位等。  
    在文件结构被释放时引用这个操作. 如同 open, release 可以为 NULL.

int(\*synch)(struct file \*,struct dentry \*,int datasync);  
刷新待处理的数据,允许进程把所有的脏缓冲区刷新到磁盘。

int (\*aio\_fsync)(struct kiocb \*, int);  
 这是 fsync 方法的异步版本.所谓的fsync方法是一个系统调用函数。系统调用fsync  
把文件所指定的文件的所有脏缓冲区写到磁盘中（如果需要，还包括存有索引节点的缓冲区）。  
相应的服务例程获得文件对象的地址，并随后调用fsync方法。通常这个方法以调用函数\_\_writeback\_single\_inode()结束，  
这个函数把与被选中的索引节点相关的脏页和索引节点本身都写回磁盘。

int (\*fasync) (int, struct file \*, int);  
这个函数是系统支持异步通知的设备驱动，下面是这个函数的模板：  
static int \*\*\*\_fasync(int fd,struct file \*filp,int mode)  
{  
    struct \*\*\*\_dev \* dev=filp->private\_data;  
    return fasync\_helper(fd,filp,mode,&dev->async\_queue);//第四个参数为 fasync\_struct结构体指针的指针。  
//这个函数是用来处理FASYNC标志的函数。（FASYNC：表示兼容BSD的fcntl同步操作）当这个标志改变时，驱动程序中的fasync（）函数将得到执行。  
}  
此操作用来通知设备它的 FASYNC 标志的改变. 异步通知是一个高级的主题, 在第 6 章中描述.  
这个成员可以是NULL 如果驱动不支持异步通知.

int (\*lock) (struct file \*, int, struct file\_lock \*);  
lock 方法用来实现文件加锁; 加锁对常规文件是必不可少的特性, 但是设备驱动几乎从不实现它.

ssize\_t (\*readv) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);  
ssize\_t (\*writev) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);  
这些方法实现发散/汇聚读和写操作. 应用程序偶尔需要做一个包含多个内存区的单个读或写操作;  
 这些系统调用允许它们这样做而不必对数据进行额外拷贝. 如果这些函数指针为 NULL, read 和 write 方法被调用( 可能多于一次 ).

ssize\_t (\*sendfile)(struct file \*, loff\_t \*, size\_t, read\_actor\_t, void \*);  
这个方法实现 sendfile 系统调用的读, 使用最少的拷贝从一个文件描述符搬移数据到另一个.  
例如, 它被一个需要发送文件内容到一个网络连接的 web 服务器使用. 设备驱动常常使 sendfile 为 NULL.

ssize\_t (\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*, int);  
sendpage 是 sendfile 的另一半; 它由内核调用来发送数据, 一次一页, 到对应的文件. 设备驱动实际上不实现 sendpage.

unsigned long (\*get\_unmapped\_area)(struct file \*, unsigned long, unsigned long, unsigned long, unsigned long);  
这个方法的目的是在进程的地址空间找一个合适的位置来映射在底层设备上的内存段中.  
这个任务通常由内存管理代码进行; 这个方法存在为了使驱动能强制特殊设备可能有的任何的对齐请求. 大部分驱动可以置这个方法为 NULL.[10]

int (\*check\_flags)(int)  
这个方法允许模块检查传递给 fnctl(F\_SETFL...) 调用的标志.

int (\*dir\_notify)(struct file \*, unsigned long);  
这个方法在应用程序使用 fcntl 来请求目录改变通知时调用. 只对文件系统有用; 驱动不需要实现 dir\_notify.

#### 3.3.3.1 实验代码

一般情况下，进行设备驱动程序的设计只是比较注重下面的几个方法：  
struct file\_operations \*\*\*\_ops={  
 .owner =  THIS\_MODULE,  
 .llseek =  \*\*\*\_llseek,  
 .read =  \*\*\*\_read,  
 .write =  \*\*\*\_write,  
 .ioctl =  \*\*\*\_ioctl,  
 .open =  \*\*\*\_open,  
 .release = \*\*\*\_release,

.mmap = \*\*\*\_mmap,  
};

原型：int (\*[mmap](http://lxr.linux.no/linux+*/+code=mmap)) (struct [file](http://lxr.linux.no/linux+*/+code=file) \*, struct [vm\_area\_struct](http://lxr.linux.no/linux+*/+code=vm_area_struct) \*);

\*\*\*\_mmap(struct file \*file, struct vm\_area\_struct \*vma);

File很好理解如何来的，但是vma是那里来的呢？

原型： int (\*[open](http://lxr.linux.no/linux+*/+code=open)) (struct [inode](http://lxr.linux.no/linux+*/+code=inode) \*, struct [file](http://lxr.linux.no/linux+*/+code=file) \*);

static int \*\*\*\_open(struct inode \*inoe, struct file \*file)

get\_user\_pages()和get\_user\_pages\_fast()的如何使用？

### 3.3.3 内存管理 miscdevice

不错的讲解：不是今天重点 <https://blog.csdn.net/u013377887/article/details/72847213>

### 3.3.3 内存管理 slab kmem\_cache

#### 3.3.3.1 kmem\_cache

struct kmem\_cache {

/\* 1) per-CPU数据，在每次分配/释放期间都会访问 \*/

struct array\_cache \*array[NR\_CPUS];

/\* 2) 可调整的缓存参数。由cache\_chain\_mutex保护 \*/

unsigned int batchcount;

unsigned int limit;

unsigned int shared;

unsigned int buffer\_size;

u32 reciprocal\_buffer\_size;

/\* 3) 后端每次分配和释放内存时都会访问 \*/

unsigned int flags; /\* 常数标志 \*/

unsigned int num; /\* 每个slab中对象的数量 \*/

/\* 4) 缓存的增长/缩减 \*/

/\* 每个slab中页数，取以2为底数的对数 \*/

unsigned int gfporder;

/\* 强制的GFP标志，例如GFP\_DMA \*/

gfp\_t gfpflags;

size\_t colour; /\* 缓存着色范围 \*/

unsigned int colour\_off; /\* 着色偏移 \*/

struct kmem\_cache \*slabp\_cache;

unsigned int slab\_size;

unsigned int dflags; /\* 动态标志 \*/

/\* 构造函数 \*/

void (\*ctor)(struct kmem\_cache \*, void \*);

/\* 5) 缓存创建/删除 \*/

const char \*name;

struct list\_head next;

/\* 6) 统计量 \*/

...

struct kmem\_list3 \*nodelists[MAX\_NUMNODES];

};

struct kmem\_list3 {

struct list\_head slabs\_partial; /\* 首先是部分空闲链表，以便生成性能更好的汇编代码 \*/

struct list\_head slabs\_full;

struct list\_head slabs\_free;

unsigned long free\_objects;

unsigned int free\_limit;

unsigned int colour\_next; /\* 各结点缓存着色 \*/

spinlock\_t list\_lock;

struct array\_cache \*shared; /\* 结点内共享 \*/

struct array\_cache \*\*alien; /\* 在其他结点上 \*/

unsigned long next\_reap; /\* 无需锁定即可更新 \*/

int free\_touched; /\* 无需锁定即可更新 \*/

};

#### 3.3.3.1 slab值得思考的问题

<https://blog.csdn.net/lukuen/article/details/6935068> 牛逼的讲解

Slab的几个问答题：

为什么Linux内存管理要用到slab分配器？

它能解决什么问题？

它的核心思想是什么？

它的最小分配单元是多大？最大分配单元又是多少？

它是否依赖buddy系统？或者和buddy系统有啥关系？

它有什么优点？又有什么缺点？

为什么要着色？着色有什么好处？又有什么不好的地方？

### 3.3.3 linux内存映射原理

为什么需要内存映射？

IO速度远远没有，读取虚拟内存的速度快。

页框号和物理内存有关系吗？

同一个虚拟地址，可以对应不同的物理地址吗？

如何查找物理内存有多少个page？每个page的地址是多少？

用户空间使用mmap()映射函数。

Mmap由用户空间映射一段文件（磁盘空间）的方法。

但是mmap的原理是什么呢？

猜测1：用户空间打开一个文件，计算出大小之后，申请页框，设置页框的属性不能被内核在使用。

虚拟内存空间如何映射文件的？

### 3.3.3 linux内存映射实现

用户空间内存-->内核空间，如何从用户空间申请物理内存？

编写一个字符驱动程序，当用户mmap映射这个物理内存到进程地址空间。

### 3.3.3 linux内存映射测试

正文

### 3.3.3 内存管理 struct zone

**include/linux/mmzone.h**

struct [**zone**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/zone) {

*/\* Read-mostly fields \*/*

*/\* zone watermarks, access with \*\_wmark\_pages(zone) macros \*/*

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) [**watermark**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/watermark)[[**NR\_WMARK**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/NR_WMARK)];

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) nr\_reserved\_highatomic;

*/\**

*\* We don't know if the memory that we're going to allocate will be*

*\* freeable or/and it will be released eventually, so to avoid totally*

*\* wasting several GB of ram we must reserve some of the lower zone*

*\* memory (otherwise we risk to run OOM on the lower zones despite*

*\* there being tons of freeable ram on the higher zones). This array is*

*\* recalculated at runtime if the sysctl\_lowmem\_reserve\_ratio sysctl*

*\* changes.*

*\*/*

[**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) lowmem\_reserve[[**MAX\_NR\_ZONES**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/MAX_NR_ZONES)];

#ifdef [**CONFIG\_NUMA**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_NUMA)

int [**node**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/node);

#endif

struct [**pglist\_data**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/pglist_data) \*zone\_pgdat;

struct [**per\_cpu\_pageset**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/per_cpu_pageset) [**\_\_percpu**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/__percpu) \*pageset;

#ifndef [**CONFIG\_SPARSEMEM**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_SPARSEMEM)

*/\**

*\* Flags for a pageblock\_nr\_pages block. See pageblock-flags.h.*

*\* In SPARSEMEM, this map is stored in struct mem\_section*

*\*/*

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) \*pageblock\_flags;

#endif */\* CONFIG\_SPARSEMEM \*/*

*/\* zone\_start\_pfn == zone\_start\_paddr >> PAGE\_SHIFT \*/*

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) zone\_start\_pfn;

*/\**

*\* spanned\_pages is the total pages spanned by the zone, including*

*\* holes, which is calculated as:*

*\* spanned\_pages = zone\_end\_pfn - zone\_start\_pfn;*

*\**

*\* present\_pages is physical pages existing within the zone, which*

*\* is calculated as:*

*\* present\_pages = spanned\_pages - absent\_pages(pages in holes);*

*\**

*\* managed\_pages is present pages managed by the buddy system, which*

*\* is calculated as (reserved\_pages includes pages allocated by the*

*\* bootmem allocator):*

*\* managed\_pages = present\_pages - reserved\_pages;*

*\**

*\* So present\_pages may be used by memory hotplug or memory power*

*\* management logic to figure out unmanaged pages by checking*

*\* (present\_pages - managed\_pages). And managed\_pages should be used*

*\* by page allocator and vm scanner to calculate all kinds of watermarks*

*\* and thresholds.*

*\**

*\* Locking rules:*

*\**

*\* zone\_start\_pfn and spanned\_pages are protected by span\_seqlock.*

*\* It is a seqlock because it has to be read outside of zone->lock,*

*\* and it is done in the main allocator path. But, it is written*

*\* quite infrequently.*

*\**

*\* The span\_seq lock is declared along with zone->lock because it is*

*\* frequently read in proximity to zone->lock. It's good to*

*\* give them a chance of being in the same cacheline.*

*\**

*\* Write access to present\_pages at runtime should be protected by*

*\* mem\_hotplug\_begin/end(). Any reader who can't tolerant drift of*

*\* present\_pages should get\_online\_mems() to get a stable value.*

*\**

*\* Read access to managed\_pages should be safe because it's unsigned*

*\* long. Write access to zone->managed\_pages and totalram\_pages are*

*\* protected by managed\_page\_count\_lock at runtime. Idealy only*

*\* adjust\_managed\_page\_count() should be used instead of directly*

*\* touching zone->managed\_pages and totalram\_pages.*

*\*/*

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) managed\_pages;

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) spanned\_pages;

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) present\_pages;

const char \*name;

#ifdef [**CONFIG\_MEMORY\_ISOLATION**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_MEMORY_ISOLATION)

*/\**

*\* Number of isolated pageblock. It is used to solve incorrect*

*\* freepage counting problem due to racy retrieving migratetype*

*\* of pageblock. Protected by zone->lock.*

*\*/*

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) nr\_isolate\_pageblock;

#endif

#ifdef [**CONFIG\_MEMORY\_HOTPLUG**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_MEMORY_HOTPLUG)

*/\* see spanned/present\_pages for more description \*/*

[**seqlock\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/seqlock_t) span\_seqlock;

#endif

int [**initialized**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/initialized);

*/\* Write-intensive fields used from the page allocator \*/*

[**ZONE\_PADDING**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/ZONE_PADDING)(\_pad1\_)

*/\* free areas of different sizes \*/*

\* MAX\_ORDER为11，分别代表包含大小为1,2,4,8,16,32,64,128,256,512,1024个连续页框的链表，具体见下面 \*/

struct [**free\_area**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/free_area) [**free\_area**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/free_area)[[**MAX\_ORDER**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/MAX_ORDER)];//max\_ORDER = 11

*/\* zone flags, see below \*/*

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) flags;

*/\* Primarily protects free\_area \*/*

[**spinlock\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/spinlock_t) lock;

*/\* Write-intensive fields used by compaction and vmstats. \*/*

[**ZONE\_PADDING**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/ZONE_PADDING)(\_pad2\_)

*/\**

*\* When free pages are below this point, additional steps are taken*

*\* when reading the number of free pages to avoid per-cpu counter*

*\* drift allowing watermarks to be breached*

*\*/*

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) percpu\_drift\_mark;

#if defined [**CONFIG\_COMPACTION**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_COMPACTION) || defined [**CONFIG\_CMA**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_CMA)

*/\* pfn where compaction free scanner should start \*/*

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) compact\_cached\_free\_pfn;

*/\* pfn where async and sync compaction migration scanner should start \*/*

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) compact\_cached\_migrate\_pfn[2];

#endif

#ifdef [**CONFIG\_COMPACTION**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_COMPACTION)

*/\**

*\* On compaction failure, 1<<compact\_defer\_shift compactions*

*\* are skipped before trying again. The number attempted since*

*\* last failure is tracked with compact\_considered.*

*\*/*

unsigned int compact\_considered;

unsigned int compact\_defer\_shift;

int compact\_order\_failed;

#endif

#if defined [**CONFIG\_COMPACTION**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_COMPACTION) || defined [**CONFIG\_CMA**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_CMA)

*/\* Set to true when the PG\_migrate\_skip bits should be cleared \*/*

[**bool**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/bool) compact\_blockskip\_flush;

#endif

[**bool**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/bool) contiguous;

[**ZONE\_PADDING**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/ZONE_PADDING)(\_pad3\_)

*/\* Zone statistics \*/*

[**atomic\_long\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/atomic_long_t) [**vm\_stat**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/vm_stat)[[**NR\_VM\_ZONE\_STAT\_ITEMS**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/NR_VM_ZONE_STAT_ITEMS)];

} [**\_\_\_\_cacheline\_internodealigned\_in\_smp**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/____cacheline_internodealigned_in_smp);

enum [**pgdat\_flags**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/pgdat_flags) {

[**PGDAT\_CONGESTED**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/PGDAT_CONGESTED), */\* pgdat has many dirty pages backed by*

*\* a congested BDI*

*\*/*

[**PGDAT\_DIRTY**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/PGDAT_DIRTY), */\* reclaim scanning has recently found*

*\* many dirty file pages at the tail*

*\* of the LRU.*

*\*/*

[**PGDAT\_WRITEBACK**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/PGDAT_WRITEBACK), */\* reclaim scanning has recently found*

*\* many pages under writeback*

*\*/*

[**PGDAT\_RECLAIM\_LOCKED**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/PGDAT_RECLAIM_LOCKED), */\* prevents concurrent reclaim \*/*

};

### 3.3.3 内存管理 struct free\_arae

**include/linux/mmzone.h**

/\* 伙伴系统的一个块，描述1,2,4,8,16,32,64,128,256,512或1024个连续页框的块 \*/

/\* 指向这个块中所有空闲小块的第一个页描述符，这些小块会按照MIGRATE\_TYPES类型存放在不同指针里 \*/

struct [**free\_area**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/free_area) {

/\* 指向这个块中所有空闲小块的第一个页描述符，这些小块会按照MIGRATE\_TYPES类型存放在不同指针里 \*/

struct [**list\_head**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/list_head) [**free\_list**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/free_list)[[**MIGRATE\_TYPES**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/MIGRATE_TYPES)];

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) nr\_free;

};

### 3.3.3 内存管理 struct kmem\_cache

**include/linux/slub\_def.h**

struct [**kmem\_cache**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/kmem_cache) {

struct [**array\_cache**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/array_cache) [**\_\_percpu**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/__percpu) \*[**cpu\_cache**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/cpu_cache);

*/\* 1) Cache tunables. Protected by slab\_mutex \*/*

unsigned int batchcount;

unsigned int [**limit**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/limit);

unsigned int [**shared**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/shared);

unsigned int size;

struct [**reciprocal\_value**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/reciprocal_value) reciprocal\_buffer\_size;

*/\* 2) touched by every alloc & free from the backend \*/*

unsigned int flags; */\* constant flags \*/*

unsigned int [**num**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/num); */\* # of objs per slab \*/*

*/\* 3) cache\_grow/shrink \*/*

*/\* order of pgs per slab (2^n) \*/*

unsigned int gfporder;

*/\* force GFP flags, e.g. GFP\_DMA \*/*

[**gfp\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/gfp_t) allocflags;

[**size\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/size_t) [**colour**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/colour); */\* cache colouring range \*/*

unsigned int colour\_off; */\* colour offset \*/*

struct [**kmem\_cache**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/kmem_cache) \*freelist\_cache;

unsigned int freelist\_size;

*/\* constructor func \*/*

void (\*[**ctor**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/ctor))(void \*[**obj**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/obj));

*/\* 4) cache creation/removal \*/*

const char \*name;

struct [**list\_head**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/list_head) list;

int [**refcount**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/refcount);

int object\_size;

int [**align**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/align);

*/\* 5) statistics \*/*

#ifdef [**CONFIG\_DEBUG\_SLAB**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_DEBUG_SLAB)

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) num\_active;

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) num\_allocations;

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) high\_mark;

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) grown;

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) reaped;

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) [**errors**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/errors);

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) max\_freeable;

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) node\_allocs;

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) node\_frees;

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) node\_overflow;

[**atomic\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/atomic_t) allochit;

[**atomic\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/atomic_t) allocmiss;

[**atomic\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/atomic_t) freehit;

[**atomic\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/atomic_t) freemiss;

#ifdef [**CONFIG\_DEBUG\_SLAB\_LEAK**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_DEBUG_SLAB_LEAK)

[**atomic\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/atomic_t) store\_user\_clean;

#endif

*/\**

*\* If debugging is enabled, then the allocator can add additional*

*\* fields and/or padding to every object. size contains the total*

*\* object size including these internal fields, the following two*

*\* variables contain the offset to the user object and its size.*

*\*/*

int [**obj\_offset**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/obj_offset);

#endif */\* CONFIG\_DEBUG\_SLAB \*/*

#ifdef [**CONFIG\_MEMCG**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_MEMCG)

struct [**memcg\_cache\_params**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/memcg_cache_params) memcg\_params;

#endif

#ifdef [**CONFIG\_KASAN**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_KASAN)

struct [**kasan\_cache**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/kasan_cache) kasan\_info;

#endif

#ifdef [**CONFIG\_SLAB\_FREELIST\_RANDOM**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_SLAB_FREELIST_RANDOM)

unsigned int \*random\_seq;

#endif

struct [**kmem\_cache\_node**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/kmem_cache_node) \*[**node**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/node)[[**MAX\_NUMNODES**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/MAX_NUMNODES)];

};

/\*\*\*\*\*\*\*/

1) cpu\_slab：一个per cpu变量，对于每个cpu来说，相当于一个本地内存缓存池。当分配内存的时候优先从本地cpu分配内存以保证cache的命中率。

2) flags：object分配掩码，例如经常使用的SLAB\_HWCACHE\_ALIGN标志位，代表创建的kmem\_cache管理的object按照硬件cache 对齐，一切都是为了速度。

3) min\_partial：限制struct kmem\_cache\_node中的partial链表slab的数量。虽说是mini\_partial，但是代码的本意告诉我这个变量是kmem\_cache\_node中partial链表最大slab数量，如果大于这个mini\_partial的值，那么多余的slab就会被释放。

4) size：分配的object size

5) object\_size：实际的object size，就是创建kmem\_cache时候传递进来的参数。和size的关系就是，size是各种地址对齐之后的大小。因此，size要大于等于object\_size。

6) offset：slub分配在管理object的时候采用的方法是：既然每个object在没有分配之前不在乎每个object中存储的内容，那么完全可以在每个object中存储下一个object内存首地址，就形成了一个单链表。很巧妙的设计。那么这个地址数据存储在object什么位置呢？offset就是存储下个object地址数据相对于这个object首地址的偏移。

7) cpu\_partial：per cpu partial中所有slab的free object的数量的最大值，超过这个值就会将所有的slab转移到kmem\_cache\_node的partial链表。

8) oo：低16位代表一个slab中所有object的数量（oo & ((1 << 16) - 1)），高16位代表一个slab管理的page数量（(2^(oo 16)) pages）。

9) max：看了代码好像就是等于oo。

10) min：当按照oo大小分配内存的时候出现内存不足就会考虑min大小方式分配。min只需要可以容纳一个object即可。

11) allocflags：从伙伴系统分配内存掩码。

12) inuse：object\_size按照word对齐之后的大小。

13) align：字节对齐大小。

14) name：sysfs文件系统显示使用。

15) list：系统有一个slab\_caches链表，所有的slab都会挂入此链表。

16) node：slab节点。在NUMA系统中，每个node都有一个struct kmem\_cache\_node数据结构。

/\*\*\*\*\*\*\*/



### 3.3.3 内存管理 struct kmem\_cache\_node

mm/slab.h

struct [**kmem\_cache\_node**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/kmem_cache_node) {

[**spinlock\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/spinlock_t) [**list\_lock**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/list_lock);

#ifdef [**CONFIG\_SLAB**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_SLAB)

struct [**list\_head**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/list_head) slabs\_partial; */\* partial list first, better asm code \*/*

struct [**list\_head**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/list_head) slabs\_full;

struct [**list\_head**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/list_head) slabs\_free;

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) num\_slabs;

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) free\_objects;

unsigned int free\_limit;

unsigned int colour\_next; */\* Per-node cache coloring \*/*

struct [**array\_cache**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/array_cache) \*[**shared**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/shared); */\* shared per node \*/*

struct [**alien\_cache**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/alien_cache) \*\*alien; */\* on other nodes \*/*

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) next\_reap; */\* updated without locking \*/*

int free\_touched; */\* updated without locking \*/*

#endif

#ifdef [**CONFIG\_SLUB**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_SLUB)

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) nr\_partial;

struct [**list\_head**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/list_head) [**partial**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/partial);

#ifdef [**CONFIG\_SLUB\_DEBUG**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_SLUB_DEBUG)

[**atomic\_long\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/atomic_long_t) nr\_slabs;

[**atomic\_long\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/atomic_long_t) total\_objects;

struct [**list\_head**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/list_head) [**full**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/full);

#endif

#endif

};

1) list\_lock：自旋锁，保护数据。

2) nr\_partial：slab节点中slab的数量。

3) partial：slab节点的slab partial链表，和struct kmem\_cache\_cpu的partial链表功能类似。



### 3.3.3 内存管理 struct kmem\_cache\_cpu

**include/linux/slub\_def.h**

struct [**kmem\_cache\_cpu**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/kmem_cache_cpu) {

void \*\*freelist; */\* Pointer to next available object \*/*

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) [**tid**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/tid); */\* Globally unique transaction id \*/*

struct page \*page; */\* The slab from which we are allocating \*/*

struct page \*[**partial**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/partial); */\* Partially allocated frozen slabs \*/*

#ifdef [**CONFIG\_SLUB\_STATS**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_SLUB_STATS)

unsigned [**stat**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/stat)[[**NR\_SLUB\_STAT\_ITEMS**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/NR_SLUB_STAT_ITEMS)];

#endif

};

1) freelist：指向下一个可用的object。

2) tid：一个神奇的数字，主要用来同步作用的。

3) page：slab内存的page指针。

4) partial：本地slab partial链表。主要是一些部分使用object的slab。



## 3.4 缺页中断或者叫做缺页异常

### 3.4.1 理清两个概念

中断

缺页中断（Page Fault）

## 3.1 关于内存的一些思考题？

### 3.3.3 slab和虚拟内存mm\_struct vm\_area\_struct如何联系起来？

# 第四章网络协议栈

## 4.1 socket的调用流程

# 第五章锁与进程间通信

## 4.1控制机制

## 4.1内核锁机制

### 4.4.1原子操作

#### 4.4.4.1 原子描述

使用方法：定于在include/asm/atomic.h文件中 使用汇编语言实现的 C语言无法实现

64位操作系统的原子定义：

typedef struct { volatile int counter; } atomic\_t;

typedef struct {

    u64 \_\_aligned(8) counter;

} atomic64\_t;

注释：u64 = long long

atomic\_t u = ATOMIC\_INT(4);//初始化一个字值U=4

atomic\_add(2,&u);

地址：<https://zhuanlan.zhihu.com/p/94462374>

例子：a++属于原子操作，但是在硬件上并非是原子操作，分为三条指令完成。

ldr r3, [r3, #0] //将存储器地址为R3+0的字数据读入寄存器R3 adds r2, r3, #1 str r2, [r3, #0] //将r2中的值存到r3+0所指定的地址中

三条指令保证顺序执行不被打断呢？如果两个线程同时操作a++,不能顺序执行代码，a的结果就无法预期。

armv7指令ldrex和strex可以解决这个问题。

使用这个指令可以都strex写数据, 先写成功之后另外一个必须重新的ldrex读数据，重新计算，在写数据。



strex除了把寄存器地址写入到地址意外，还可以返回写是否成功的

#### 4.4.4.1 原子源码API

**<asm-arch/atomic.h>**typedef struct { volatile int counter; } atomic\_t;

atomic\_read(atomic\_t \*v) 读取原子变量的值  
atomic\_set(atomic\_t \*v, int i) 将v设置为i  
atomic\_add(int i, atomic\_t \*v) 将i加到v  
atomic\_add\_return(int i, atomic\_t \*v) 将i加到v，并返回结果  
atomic\_sub(int i, atomic\_t \*v) 从v减去i  
atomic\_sub\_return(int i, atomic\_t \*v) 从v减去i，并返回结果  
atomic\_sub\_and\_test(int i, atomic\_t \*v) 从v减去v。如果结果为0则返回true，否则返回false  
atomic\_inc(atomic\_t \*v) 将v加1  
atomic\_inc\_and\_test(atomic\_t \*v) 将v加1。如果结果为0则返回true，否则返回false  
atomic\_dec(atomic\_t \*v) 从v减去1  
atomic\_dec\_and\_test(atomic\_t \*v) 从v减去1。如果结果为0则返回true，否则返回false  
atomic\_add\_negative(int i, atomic\_t \*v) 将i加到v。如果结果小于0则返回true，否则返回false  
atomic\_add\_negative(int i, atomic\_t \*v) 将i加到v。如果结果为负则返回true，否则返回false

### 4.4.1自旋锁

#### 4.4.4.1 自旋锁描述

自旋锁的实现与体系结构有关，保护比较短的代码。用于死等模式，不能处理临界区比加大的代码。还要注意一点，自选锁定义的临界区不能出现会休眠的代码。原因在于（个人理解：因为进入临界区代码之后，CPU处于不工作状态，如果出现休眠，无法解锁处于，处于死锁状态），开启自旋锁相当与关闭了本CPU内核抢占。

PS: kmalloc()属于可以休眠函数，使用需要注意

#### 4.4.4.1 自旋锁API

1. 就是定义了一个变量的整数  
   typedef struct spinlock {
2. struct raw\_spinlock rlock;
3. } spinlock\_t;
5. typedef struct raw\_spinlock {
6. arch\_spinlock\_t raw\_lock;
7. } raw\_spinlock\_t;

spinlock lock;//定义锁机制

spin\_lock\_init(&lock)//初始化锁为0

spin\_lock(&lock)//加锁

//临界区代码

spin\_unlock(&lock)//解锁

自旋锁的变体

|  |  |  |
| --- | --- | --- |
| 接口API的类型 | spinlock中的定义 | raw\_spinlock的定义 |
| 定义spin lock并初始化 | DEFINE\_SPINLOCK | DEFINE\_RAW\_SPINLOCK |
| 动态初始化spin lock | spin\_lock\_init | raw\_spin\_lock\_init |
| 获取指定的spin lock | spin\_lock | raw\_spin\_lock |
| 获取指定的spin lock同时disable本CPU中断 | spin\_lock\_irq | raw\_spin\_lock\_irq |
| 保存本CPU当前的irq状态，disable本CPU中断并获取指定的spin lock | spin\_lock\_irqsave | raw\_spin\_lock\_irqsave |
| 获取指定的spin lock同时disable本CPU的bottom half | spin\_lock\_bh | raw\_spin\_lock\_bh |
| 释放指定的spin lock | spin\_unlock | raw\_spin\_unlock |
| 释放指定的spin lock同时enable本CPU中断 | spin\_unlock\_irq | raw\_spin\_unock\_irq |
| 释放指定的spin lock同时恢复本CPU的中断状态 | spin\_unlock\_irqstore | raw\_spin\_unlock\_irqstore |
| 获取指定的spin lock同时enable本CPU的bottom half | spin\_unlock\_bh | raw\_spin\_unlock\_bh |
| 尝试去获取spin lock，如果失败，不会spin，而是返回非零值 | spin\_trylock | raw\_spin\_trylock |
| 判断spin lock是否是locked，如果其他的thread已经获取了该lock，那么返回非零值，否则返回0 |  |  |

### 4.4.1信号量&互斥量

#### 4.4.4.1 信号量介绍

个人的一些见解：

Down\_read(),对临界区操作，如果在临界区之间有修改代码参数怎能办？

内核的信号量同时允许多个读，只允许一个写。

问题是为什需要down\_read()和up\_read()，直接直接定义一个volatile一个变量，不是就可以了吗？

内核使用的信号量定义如下。用户空间信号量的实现有所不同，将在5.3.2节讲述。  
**<asm-arch/semaphore.h>**struct semaphore {  
atomic\_t count;  
int sleepers;  
wait\_queue\_head\_t wait;  
};  
尽管该结构定义在体系结构相关的头文件中，但大多数体系结构都使用了这里给出的结构。  
❑count指定了可以同时处于信号量保护的临界区中进程的数目。 count == 1用于大多数情况  
（此类信号量又名互斥信号量，因为它们用于实现互斥）。  
❑sleepers指定了等待允许进入临界区的进程的数目。不同于自旋锁，等待的进程会进入睡眠  
状态，直至信号量释放才会被唤醒。这意味着相关的CPU在同时可以执行其他任务。  
❑wait用于实现一个队列，保存所有在该信号量上睡眠的进程的task\_struct（第14章讲述了  
相关的底层机制）。

#### 4.4.4.1 信号量代码

信号量代码

Struct semaphore my\_sem;

Sema\_init(&my\_sem,5);//初始化5个信号

down(&my\_sem);//占用资源

//临界区代码 比较大

up(&my\_sem);//释放资源

互斥量代码

struct mutex my\_mutex;数据类型，

mutex\_lock(&my\_mutex)//加锁

//临界区代码

mutex\_unlock(&my\_mutex)//解锁

信号量（semaphore）和互斥量（mutex）之间的区别？

信号量的占用与释放可以不同进程操作，互斥量加锁与解锁必须同一进程

信号量用于同步（保持一定的访问顺序）

互斥量用于互斥（会处于休眠的）

假设一种场景，对于一个资源只允许同时一个写多个读，互斥量不能满足，互斥量最根本的是用于对资源的保护。

### 4.4.1 RCU

#### 4.4.4.1 RCU描述

Read-Copy Update

#### 4.4.4.1 RCUAPI

### 4.4.1内存和优化屏障

#### 4.4.4.1 内存和优化屏障描述

为什么需要内存屏障，为了提高计算机性能，（编译器优化和处理器的优化）指令重排不影响结果。

#### 4.4.4.1 内存和优化屏障API

### 4.4.1读者与写着锁

#### 4.4.4.1 读者与写着锁描述

#### 4.4.4.1 读者与写着锁API

### 4.4.1大内核锁

#### 4.4.4.1 大内核锁描述

#### 4.4.4.1 大内核锁API

## 4.1 用户进程通信

### 4.4.1 System V

#### 4.4.4.1 信号量

信号量就是一个PV操作，当资源数量是1，就是一个互斥信号量

#include <semaphore.h>

int ar[10] = {1,2,3,4,5,6,7,8,9,10};

sem\_t s;//定义一个信号量集合

void \*do\_bank(void \*arg)

{

int i;

//P(s);

sem\_wait(&s);//P操作减少一个窗口被占用

printf("[%d]starting....\n",(int )arg);

for(i =0 ;i<5 ;i++)

{

fflush(stdout);//强制刷新

sleep(1);

}

printf("done.\n");

sem\_post(&s);//V操作 使用完毕归还资源

//V(s);

}

int main()

{

int res,i;

res = sem\_init(&s,0,2);//设置3个共享资源 三个窗口可以工作

pthread\_t t[5];

void \*thread\_result;

for(i=0 ;i<10 ;i++)

{

res = pthread\_create(&t[i] , NULL, do\_bank ,(void \*)i);

}

for(i=0 ;i<10 ;i++)

{

pthread\_join(t[i],NULL);

}

sem\_destroy(&s);//销毁信号量

return 0;

}

#### 4.4.4.1 消息队列

https://blog.csdn.net/zhourong0511/article/details/80143465 牛逼引用讲解

##一、消息队列 ##

——想知道如何通过消息队列实现进程间通信，我们必须得了解三个概念。

1. 消息队列

2. 消息缓冲区

3. 通道



我们来参照这个图分析一下：

什么是消息对列？

——就是一个数据结构，是一个队列。主要是用来实现消息传递。（可以理解为一个链式结构）

什么是消息缓存区？

——如上图，那些绿块块分别是一个一个的消息缓存区。用来存放通道号，和你写入通道中的数据。

struct msgbuf{

long channel; //通道号，必须大于0（系统内部其实将它定义为 mtype ——把它视为一个类型）

char mtext[100];//消息内容（100是自定义的值，你可以任意更改）

}:

什么是通道？

——通道号就相当于一个分类，并不真实存在。channel 值相同的属于同一通道，系统可以根据通道号来选择对应的消息队列。

联系实例分析：就好比看电视，不同的频道对应不同的通道，想看哪个频道，就从哪个通道进入

在写数据时，对通道号，也是有一定的要求的：

channel=0，——非法（报错）

channel<0，——非法（报错）

channel>0，——可以，继续写入消息内容

在读数据时，对通道号，也是有一定的要求的：

channel=0，——返回队列的第一个消息

channel>0，——返回队列中该通道号的第一个消息（相同的通道号归为一类，一个通道里可以放很多相同的通道号）

channel<0，——返回队列中你输入通道号的绝对值，所对应的通道号（若有多个相同的通道号，默认存放最小消息内容的通道号）

##二、 对消息队列的各种操作 ##

1.查看消息队列：ipcs -q

此时消息队列已经创建：



##2. 删除消息队列：ipcrm -Q key ##

3.创建/打开消息队列：

#include<sys/ipc.h>

#include<sys/msg.h>

#include<sys/types.h>

//用于创建一个新的或者打开一个已存在的消息队列

int msgget(key\_t key, int flag);

//返回值：-1（失败） 成功（你创建/打开的消息队列的标识符id）

代码实现：

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int main( void )

{

int id = msgget(1234, IPC\_CREAT|0644);

if ( id == -1 ) perror("msgget"),exit(1);

printf("create ok\n");

}



解释msgget参数 ：##

（1）key（关键字）

每个消息队列都有一个唯一的标识符id，可以就是消息队列的关键字，根据key的信息，找到对应的消息队列

判断一个消息队列是否存在，直接用key与已有的消息队列关键字比较，若key值相同，说明已存在，key值不同，说明没有这个消息队列，可新创建

（2）flag（权限）

IPC\_CREAT——创建一个新的消息队列

IPC\_EXCL——表示创建的消息队列存在，返回错误（与IPC\_CREATE一同使用）

0 ——是已存在的消息队列，直接打开

IPC\_NOWAIT——读写消息队列无法满足时，不阻塞。（是msgsnd 和msgrcv 的权限）

（3）key的生成

生成key，需要一个函数——ftok（）

key\_t ftok(const char\*filename,proj\_id)

第一个参数：

//filename是一个文件名（路径），要求这个文件名必须存在——（一般用const修饰）

//一般默认为当前文件

例：key\_t key;

key=(".",1)——将当前目录设为你的文件名

第二个参数：是子序号，虽然是int型，但是只使用8 bits，（1~255）

返回值：-1（失败） （成功）key\_t 的结果

代码实现：

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<string.h>

#include<sys/ipc.h>

#include<sys/msg.h>

#include <sys/types.h>

int main()

{

key\_t key=ftok(".",'a');

if(key == -1)

perror("ftok"),exit(1);

int id=msgget(key,IPC\_CREAT|0644);

if(id == -1)

perror("msgget"),exit(1);

printf("create success\n");

return 0;

}

4.往消息队列里写数据

int msgsnd（int misqid,const void\*msgp,size\_t msgsz,int msgflg）

//misgid——msgget函数的返回值,想写入内容的消息队列的标识符id

//msgp——临时创建一个消息队列结构体对象的指针

两者之间的关系：

1.先将想写入的消息内容存放在 消息队列结构体的对象中，这个对象相当于一个中转站;

2.再将中转站里的内容放入到 miqid对应的消息队列中；

3.从而间接的实现 往消息队列中写数据。

//mgsz——写入消息的大小，char metex[100]的大小，不包括通道号

//msgflg——权限（0——阻塞方式，当输入的通道号不合法，会一直阻塞，不会结束，除非用Ctrl C

IPC\_NOWAIT——非阻塞方式，当输入的通道号不合法，不会阻塞，直接结束。

代码实现：

#include <stdio.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/ipc.h>

#include <sys/msg.h>

struct msgbuf {

long channel;

char mtext[100];

};

int main( void )

{

int id = msgget(0x000004d2,0);

//打开消息队列，0x000004d2是刚才创建的消息队列的关键字

//第二个参数为0，说明该消息队列已经存在，这里进行的是打开操作而不是创建

if ( id == -1 ) perror("msgget"),exit(1);

struct msgbuf mb;

memset(&mb,0,sizeof(mb));//一定要记得初始化，否则因为缓冲区的问题，会出现错误

printf("channel:");

scanf("%ld", &mb.channel);

printf("text:");

scanf("%s", mb.mtext);

int r = msgsnd(id, &mb, strlen(mb.mtext), 0);

if ( r == -1 ) perror("msgsnd"),exit(1);

}

5.从消息队列中读数据：

int msgrcv(int msggid,void\*msgp,size\_t msgsz,int chnnel,int msgflg)

//misgid——msgget函数的返回值,想写入内容的消息队列的标识符id

//msgp——临时创建一个消息队列结构体对象的指针

两者之间的关系：

1.先将msgqid对应的消息队列的内容放在这个中转站（临时创建的结构体对象）中;

2.再对中转站里的内容进行读取；

3.从而间接的实现 从消息队列中读数据。

//mgsz——写入消息的大小，char metex[100]的大小，不包括通道号

//msgflg——权限（0——阻塞方式，当输入的通道号不合法，会一直阻塞，不会结束，除非用Ctrl C

IPC\_NOWAIT——非阻塞方式，当输入的通道号不合法，不会阻塞，直接结束。

1

2

3

4

5

6

7

8

9

10

11

12

13

代码实现：

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/ipc.h>

#include<sys/msg.h>

#include<sys/types.h>

struct msgbuf{

long channel;

char mtext[100];

};

int main(void)

{

int id=msgget(0x000004d2,0);

//打开消息队列，0x000004d2是刚才创建的消息队列的关键字

//第二个can顺为0，说明该消息队列已经存在，这里进行的是打开操作而不是创建

if(id==-1)

perror("msgget"),exit(1);

struct msgbuf mb;

memset(&mb,0,sizeof(mb));//一定要记得初始化，否则因为缓冲区的问题，会出现错误

printf("channel:");

int channel;

scanf("%d",&channel);

ssize\_t r= msgrcv(id,&mb,100,channel,0);

if(r==-1)

perror("msgrcv"),exit(1);

printf("%s\n",mb.mtext);

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

6.删除消息队列：

#include <sys/ipc.h>

#include <sys/msg.h>

#include<sys/types.h>

int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

//返回值：成功返回0，失败返回-1

//msqid 消息队列标识符id，mspid是msgget的返回值

//cmd：包括以下三种

1. IPC\_STAT

读取消息队列的数据结构msqid\_ds，并将其存储在buf指定的地址中。

2.IPC\_SET

设置消息队列的数据结构msqid\_ds中的ipc\_perm元素的值。这个值取自buf参数。

3.IPC\_RMID

从系统内核中移走消息队列，即删除消息队列

(buf 配合IPC\_RMID使用，设为0，即buf为NULL)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

代码实现：

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<string.h>

#include<sys/ipc.h>

#include<sys/msg.h>

#include<sys/types.h>

int main()

{

int id=msgget(111,IPC\_CREAT|0644);

if(id == -1)

perror("msgget"),exit(1);

printf("create success\n");

if(msgctl(id,IPC\_RMID,0) == -1)

perror("msgtcl"),exit(1);

printf("msgctl success\n");

return 0;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

————————————————

版权声明：本文为CSDN博主「有梦想的-蜗牛」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/zhourong0511/article/details/80143465

#### 4.4.4.1 共享内存

<https://www.cnblogs.com/wuyepeng/p/9748889.html> 实验带讲解比较牛逼。

引用文章：

1.什么是共享内存？

共享内存就是允许两个或多个进程共享一定的存储区。就如同 malloc() 函数向不同进程返回了指向同一个物理内存区域的指针。当一个进程改变了这块地址中的内容的时候，其它进程都会察觉到这个更改。因为数据不需要在客户机和服务器端之间复制，数据直接写到内存，不用若干次数据拷贝，所以这是最快的一种IPC。

注：共享内存没有任何的同步与互斥机制，所以要使用信号量来实现对共享内存的存取的同步。



**共享内存特点和优势**

当中共享内存的大致原理相信我们可以看明白了，就是让两个进程地址通过页表映射到同一片物理地址以便于通信,你可以给一个区域里面写入数据，理所当然你就可以从中拿取数据，这也就构成了进程间的双向通信，而且共享内存是IPC通信当中**传输速度最快的通信方式没有之一**，理由很简单，客户进程和服务进程传递的数据直接从内存里存取、放入，数据不需要在两进程间复制，没有什么操作比这简单了。再者用共享内存进行数据通信，它对数据也没啥限制。

最后就是共享内存的生命周期随内核。即所有访问共享内存区域对象的进程都已经正常结束,共享内存区域对象仍然在内核中存在(除非显式删除共享内存区域对象),在内核重新引导之前,对该共享内存区域对象的任何改写操作都将一直保留;简单地说,共享内存区域对象的生命周期跟系统内核的生命周期是一致的,而且共享内存区域对象的作用域范围就是在整个系统内核的生命周期之内。

**缺陷**

但是，共享内存也并不完美，共享内存并未提供同步机制，也就是说，在一个服务进程结束对共享内存的写操作之前，并没有自动机制可以阻止另一个进程（客户进程）开始对它进行读取。这明显还达不到我们想要的，我们不单是在两进程间交互数据，还想实现多个进程对共享内存的同步访问，这也正是使用共享内存的窍门所在。基于此，我们通常会用平时常谈到和用到 **信号量**来实现对共享内存同步访问控制。

与共享内存有关的函数

所有的函数共用头文件

|  |  |
| --- | --- |
| 1  2  3 | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/shm.h> |

　　创建共享内存——>shmget() 函数

|  |  |
| --- | --- |
| 1  2 | int shmget(key\_t key, size\_t size, int shmflg);                  //成功返回共享内存的ID,出错返回-1 |

(1)第一个参数key是长整型（唯一非零），系统建立IPC通讯 （ 消息队列、 信号量和 共享内存） 时必须指定一个ID值。通常情况下，该id值通过ftok函数得到，由内核变成标识符，要想让两个进程看到同一个信号集，只需设置key值不变就可以。

 (2)第二个参数size指定共享内存的大小，它的值一般为一页大小的整数倍（未到一页，操作系统向上对齐到一页，但是用户实际能使用只有自己所申请的大小）。

 (3)第三个参数shmflg是一组标志，创建一个新的共享内存，将shmflg 设置了IPC\_CREAT标志后，共享内存存在就打开。而IPC\_CREAT | IPC\_EXCL则可以创建一个新的，唯一的共享内存，如果共享内存已存在，返回一个错误。一般我们会还或上一个文件权限

3.2操作共享内存———>shmctl()函数

|  |  |
| --- | --- |
| 1  2 | int shmctl(int shm\_id, int cmd, struct shmid\_ds \*buf);                  //成功返回0，出错返回-1 |

(1)第一个参数，shm\_id是shmget函数返回的共享内存标识符。

(2)第二个参数，cmd是要采取的操作，它可以取下面的三个值 ：

IPC\_STAT：把shmid\_ds结构中的数据设置为共享内存的当前关联值，即用共享内存的当前关联值覆盖shmid\_ds的值。

IPC\_SET：如果进程有足够的权限，就把共享内存的当前关联值设置为shmid\_ds结构中给出的值

IPC\_RMID：删除共享内存段

(3)第三个参数，buf是一个结构指针，它指向共享内存模式和访问权限的结构。 shmid\_ds结构至少包括以下成员

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | struct shmid\_ds  {      uid\_t shm\_perm.uid;      uid\_t shm\_perm.gid;      mode\_t shm\_perm.mode;  }; |

挂接操作———>shmat()函数

创建共享存储段之后，将进程连接到它的地址空间

|  |  |
| --- | --- |
| 1  2 | void \*shmat(int shm\_id, const void \*shm\_addr, int shmflg);                      //成功返回指向共享存储段的指针，出错返回-1 |

(1)第一个参数，shm\_id是由shmget函数返回的共享内存标识。

(2)第二个参数，shm\_addr指定共享内存连接到当前进程中的地址位置，通常为空，表示让系统来选择共享内存的地址。

(3)第三个参数，shm\_flg是一组标志位，通常为0

3.4分离操作———>shmdt()函数

该操作不从系统中删除标识符和其数据结构，要显示调用shmctl(带命令IPC\_RMID)才能删除它

|  |  |
| --- | --- |
| 1  2 | int shmdt(const void \*shmaddr);              //成功返回0，出错返回-1 |

(1)addr参数是以前调用shmat时的返回值

4.模拟实现进程间的通信方式———>共享内存

三、使用共享内存进行进程间通信

说了这么多，又到了实战的时候了。下面就以两个不相关的进程来说明进程间如何通过共享内存来进行通信。其中一个文件shmread.c创建共享内存，并读取其中的信息，另一个文件shmwrite.c向共享内存中写入数据。为了方便操作和数据结构的统一，为这两个文件定义了相同的数据结构，定义在文件shmdata.c中。结构shared\_use\_st中的written作为一个可读或可写的标志，非0：表示可读，0表示可写，text则是内存中的文件。

shmdata.h的源代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #ifndef \_SHMDATA\_H\_HEADER  #define \_SHMDATA\_H\_HEADER  #define TEXT\_SZ 2048  struct shared\_use\_st  {      int written;//作为一个标志，非0：表示可读，0表示可写      char text[TEXT\_SZ];//记录写入和读取的文本  };  #endif |

　　源文件shmread.c的源代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | #include <unistd.h>  #include <stdlib.h>  #include <stdio.h>  #include <sys/shm.h>  #include "shmdata.h"  int main()  {      int running = 1;//程序是否继续运行的标志      void \*shm = NULL;//分配的共享内存的原始首地址      struct shared\_use\_st \*shared;//指向shm      int shmid;//共享内存标识符 //创建共享内存      shmid = shmget((key\_t)1234, sizeof(struct shared\_use\_st), 0666|IPC\_CREAT);      if(shmid == -1)      {          fprintf(stderr, "shmget failed\n");          exit(EXIT\_FAILURE);      }   //将共享内存连接到当前进程的地址空间      shm = shmat(shmid, 0, 0);      if(shm == (void\*)-1)      {          fprintf(stderr, "shmat failed\n");          exit(EXIT\_FAILURE);      }      printf("\nMemory attached at %X\n", (int)shm);  //设置共享内存      shared = (struct shared\_use\_st\*)shm;      shared->written = 0;      while(running)//读取共享内存中的数据      {       //没有进程向共享内存定数据有数据可读取          if(shared->written != 0)          {              printf("You wrote: %s", shared->text);              sleep(rand() % 3);          //读取完数据，设置written使共享内存段可写              shared->written = 0;         //输入了end，退出循环（程序）              if(strncmp(shared->text, "end", 3) == 0)                  running = 0;          }          else//有其他进程在写数据，不能读取数据              sleep(1);      }   //把共享内存从当前进程中分离      if(shmdt(shm) == -1)      {          fprintf(stderr, "shmdt failed\n");          exit(EXIT\_FAILURE);      }   //删除共享内存      if(shmctl(shmid, IPC\_RMID, 0) == -1)      {          fprintf(stderr, "shmctl(IPC\_RMID) failed\n");          exit(EXIT\_FAILURE);      }      exit(EXIT\_SUCCESS);  } |

　　源文件shmwrite.c的源代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | #include <unistd.h>  #include <stdlib.h>  #include <stdio.h>  #include <string.h>  #include <sys/shm.h>  #include "shmdata.h"  int main()  {      int running = 1;      void \*shm = NULL;      struct shared\_use\_st \*shared = NULL;      char buffer[BUFSIZ + 1];//用于保存输入的文本      int shmid;  //创建共享内存      shmid = shmget((key\_t)1234, sizeof(struct shared\_use\_st), 0666|IPC\_CREAT);      if(shmid == -1)      {          fprintf(stderr, "shmget failed\n");          exit(EXIT\_FAILURE);      }   //将共享内存连接到当前进程的地址空间      shm = shmat(shmid, (void\*)0, 0);      if(shm == (void\*)-1)      {          fprintf(stderr, "shmat failed\n");          exit(EXIT\_FAILURE);      }      printf("Memory attached at %X\n", (int)shm);    //设置共享内存      shared = (struct shared\_use\_st\*)shm;      while(running)//向共享内存中写数据      {       //数据还没有被读取，则等待数据被读取,不能向共享内存中写入文本          while(shared->written == 1)          {              sleep(1);              printf("Waiting...\n");          }       //向共享内存中写入数据          printf("Enter some text: ");          fgets(buffer, BUFSIZ, stdin);          strncpy(shared->text, buffer, TEXT\_SZ);      //写完数据，设置written使共享内存段可读          shared->written = 1;     //输入了end，退出循环（程序）          if(strncmp(buffer, "end", 3) == 0)              running = 0;      }   //把共享内存从当前进程中分离      if(shmdt(shm) == -1)      {          fprintf(stderr, "shmdt failed\n");          exit(EXIT\_FAILURE);      }      sleep(2);      exit(EXIT\_SUCCESS);  } |

　　结果截图如下：





分析：

1、程序shmread创建共享内存，然后将它连接到自己的地址空间。在共享内存的开始处使用了一个结构struct\_use\_st。该结构中有个标志written，当共享内存中有其他进程向它写入数据时，共享内存中的written被设置为0，程序等待。当它不为0时，表示没有进程对共享内存写入数据，程序就从共享内存中读取数据并输出，然后重置设置共享内存中的written为0，即让其可被shmwrite进程写入数据。

2、程序shmwrite取得共享内存并连接到自己的地址空间中。检查共享内存中的written，是否为0，若不是，表示共享内存中的数据还没有被完，则等待其他进程读取完成，并提示用户等待。若共享内存的written为0，表示没有其他进程对共享内存进行读取，则提示用户输入文本，并再次设置共享内存中的written为1，表示写完成，其他进程可对共享内存进行读操作。

四、关于前面的例子的安全性讨论

这个程序是不安全的，当有多个程序同时向共享内存中读写数据时，问题就会出现。可能你会认为，可以改变一下written的使用方式，例如，只有当written为0时进程才可以向共享内存写入数据，而当一个进程只有在written不为0时才能对其进行读取，同时把written进行加1操作，读取完后进行减1操作。这就有点像文件锁中的读写锁的功能。咋看之下，它似乎能行得通。但是这都不是原子操作，所以这种做法是行不能的。试想当written为0时，如果有两个进程同时访问共享内存，它们就会发现written为0，于是两个进程都对其进行写操作，显然不行。当written为1时，有两个进程同时对共享内存进行读操作时也是如些，当这两个进程都读取完是，written就变成了-1.

要想让程序安全地执行，就要有一种进程同步的进制，保证在进入临界区的操作是原子操作。例如，可以使用前面所讲的信号量来进行进程的同步。因为信号量的操作都是原子性的。

五、使用共享内存的优缺点

1、优点：我们可以看到使用共享内存进行进程间的通信真的是非常方便，而且函数的接口也简单，数据的共享还使进程间的数据不用传送，而是直接访问内存，也加快了程序的效率。同时，它也不像匿名管道那样要求通信的进程有一定的父子关系。

2、缺点：共享内存没有提供同步的机制，这使得我们在使用共享内存进行进程间通信时，往往要借助其他的手段来进行进程间的同步工作。

### 4.4.1其它IPC机制

#### 4.4.4.1 信号(异步通信)

##### 4.4.4.1.1 信号结构体

<https://www.cnblogs.com/electronic/p/10939769.html>

https://blog.csdn.net/ypt523/article/details/80290208

信号与信号量通信是不一样的。

一个问题：

ctrl+c为什么会中断程序？

Kill -9 pid 为什么会中断程序？

答：ctrl+C 属于规定的 “9”信号，难道说，进程在不停的捕获信号吗？这不浪费CPU吗？

信号机制属于一种软件层次的一种中断。比如来了一个sigint(ctrl+c)被记录在PCB中（个人认为就是一个task\_struct中）

代码：

1. 对2号信号捕捉然后回调  
   #include<stdio.h>
2. #include<signal.h>
4. void handler()
5. {
6. printf("handler\n");
7. }
9. int main()
10. {
11. signal(2,handler);
12. while(1);
13. return 0;
14. ｝

#### 4.4.4.1 管道

#### 4.4.4.1 套接字

就是socket

## 4.1用户和内核通信机制

### 4.4.1系统调用

#### 4.4.4.1 自己如何添加一个系统调用

#### 4.4.4.1 系统调用汇总

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| https://blog.csdn.net/sinat\_26227857/article/details/44244433 | | | |  |
| 系统调用号(%eax) | 函数名 | 入口点 | 源代码 |  |
| 0 | read | sys\_read | fs/read\_write.c |  |
| 1 | write | sys\_write | fs/read\_write.c |  |
| 2 | open | sys\_open | fs/open.c |  |
| 3 | close | sys\_close | fs/open.c |  |
| 4 | stat | sys\_newstat | fs/stat.c |  |
| 5 | fstat | sys\_newfstat | fs/stat.c |  |
| 6 | lstat | sys\_newlstat | fs/stat.c |  |
| 7 | poll | sys\_poll | fs/select.c |  |
| 8 | lseek | sys\_lseek | fs/read\_write.c |  |
| 9 | mmap | sys\_mmap | arch/x86/kernel/sys\_x86\_64.c |  |
| 10 | mprotect | sys\_mproadbtect | mm/mprotect.c |  |
| 11 | munmap | sys\_munmap | mm/mmap.c |  |
| 12 | brk | sys\_brk | mm/mmap.c |  |
| 13 | rt\_sigaction | sys\_rt\_sigaction | kernel/signal.c |  |
| 14 | rt\_sigprocmask | sys\_rt\_sigprocmask | kernel/signal.c |  |
| 15 | rt\_sigreturn | stub\_rt\_sigreturn | arch/x86/kernel/signal.c |  |
| 16 | ioctl | sys\_ioctl | fs/ioctl.c |  |
| 17 | pread64 | sys\_pread64 | fs/read\_write.c |  |
| 18 | pwrite64 | sys\_pwrite64 | fs/read\_write.c |  |
| 19 | readv | sys\_readv | fs/read\_write.c |  |
| 20 | writev | sys\_writev | fs/read\_write.c |  |
| 21 | access | sys\_access | fs/open.c |  |
| 22 | pipe | sys\_pipe | fs/pipe.c |  |
| 23 | select | sys\_select | fs/select.c |  |
| 24 | sched\_yield | sys\_sched\_yield | kernel/sched/core.c |  |
| 25 | mremap | sys\_mremap | mm/mmap.c |  |
| 26 | msync | sys\_msync | mm/msync.c |  |
| 27 | mincore | sys\_mincore | mm/mincore.c |  |
| 28 | madvise | sys\_madvise | mm/madvise.c |  |
| 29 | shmget | sys\_shmget | ipc/shm.c |  |
| 30 | shmat | sys\_shmat | ipc/shm.c |  |
| 31 | shmctl | sys\_shmctl | ipc/shm.c |  |
| 32 | dup | sys\_dup | fs/file.c |  |
| 33 | dup2 | sys\_dup2 | fs/file.c |  |
| 34 | pause | sys\_pause | kernel/signal.c |  |
| 35 | nanosleep | sys\_nanosleep | kernel/hrtimer.c |  |
| 36 | getitimer | sys\_getitimer | kernel/itimer.c |  |
| 37 | alarm | sys\_alarm | kernel/timer.c |  |
| 38 | setitimer | sys\_setitimer | kernel/itimer.c |  |
| 39 | getpid | sys\_getpid | kernel/sys.c |  |
| 40 | sendfile | sys\_sendfile64 | fs/read\_write.c |  |
| 41 | socket | sys\_socket | net/socket.c |  |
| 42 | connect | sys\_connect | net/socket.c |  |
| 43 | accept | sys\_accept | net/socket.c |  |
| 44 | sendto | sys\_sendto | net/socket.c |  |
| 45 | recvfrom | sys\_recvfrom | net/socket.c |  |
| 46 | sendmsg | sys\_sendmsg | net/socket.c |  |
| 47 | recvmsg | sys\_recvmsg | net/socket.c |  |
| 48 | shutdown | sys\_shutdown | net/socket.c |  |
| 49 | bind | sys\_bind | net/socket.c |  |
| 50 | listen | sys\_listen | net/socket.c |  |
| 51 | getsockname | sys\_getsockname | net/socket.c |  |
| 52 | getpeername | sys\_getpeername | net/socket.c |  |
| 53 | socketpair | sys\_socketpair | net/socket.c |  |
| 54 | setsockopt | sys\_setsockopt | net/socket.c |  |
| 55 | getsockopt | sys\_getsockopt | net/socket.c |  |
| 56 | clone | stub\_clone | kernel/fork.c |  |
| 57 | fork | stub\_fork | kernel/fork.c |  |
| 58 | vfork | stub\_vfork | kernel/fork.c |  |
| 59 | execve | stub\_execve | fs/exec.c |  |
| 60 | exit | sys\_exit | kernel/exit.c |  |
| 61 | wait4 | sys\_wait4 | kernel/exit.c |  |
| 62 | kill | sys\_kill | kernel/signal.c |  |
| 63 | uname | sys\_newuname | kernel/sys.c |  |
| 64 | semget | sys\_semget | ipc/sem.c |  |
| 65 | semop | sys\_semop | ipc/sem.c |  |
| 66 | semctl | sys\_semctl | ipc/sem.c |  |
| 67 | shmdt | sys\_shmdt | ipc/shm.c |  |
| 68 | msgget | sys\_msgget | ipc/msg.c |  |
| 69 | msgsnd | sys\_msgsnd | ipc/msg.c |  |
| 70 | msgrcv | sys\_msgrcv | ipc/msg.c |  |
| 71 | msgctl | sys\_msgctl | ipc/msg.c |  |
| 72 | fcntl | sys\_fcntl | fs/fcntl.c |  |
| 73 | flock | sys\_flock | fs/locks.c |  |
| 74 | fsync | sys\_fsync | fs/sync.c |  |
| 75 | fdatasync | sys\_fdatasync | fs/sync.c |  |
| 76 | truncate | sys\_truncate | fs/open.c |  |
| 77 | ftruncate | sys\_ftruncate | fs/open.c |  |
| 78 | getdents | sys\_getdents | fs/readdir.c |  |
| 79 | getcwd | sys\_getcwd | fs/dcache.c |  |
| 80 | chdir | sys\_chdir | fs/open.c |  |
| 81 | fchdir | sys\_fchdir | fs/open.c |  |
| 82 | rename | sys\_rename | fs/namei.c |  |
| 83 | mkdir | sys\_mkdir | fs/namei.c |  |
| 84 | rmdir | sys\_rmdir | fs/namei.c |  |
| 85 | creat | sys\_creat | fs/open.c |  |
| 86 | link | sys\_link | fs/namei.c |  |
| 87 | unlink | sys\_unlink | fs/namei.c |  |
| 88 | symlink | sys\_symlink | fs/namei.c |  |
| 89 | readlink | sys\_readlink | fs/stat.c |  |
| 90 | chmod | sys\_chmod | fs/open.c |  |
| 91 | fchmod | sys\_fchmod | fs/open.c |  |
| 92 | chown | sys\_chown | fs/open.c |  |
| 93 | fchown | sys\_fchown | fs/open.c |  |
| 94 | lchown | sys\_lchown | fs/open.c |  |
| 95 | umask | sys\_umask | kernel/sys.c |  |
| 96 | gettimeofday | sys\_gettimeofday | kernel/time.c |  |
| 97 | getrlimit | sys\_getrlimit | kernel/sys.c |  |
| 98 | getrusage | sys\_getrusage | kernel/sys.c |  |
| 99 | sysinfo | sys\_sysinfo | kernel/sys.c |  |
| 100 | times | sys\_times | kernel/sys.c |  |
| 101 | ptrace | sys\_ptrace | kernel/ptrace.c |  |
| 102 | getuid | sys\_getuid | kernel/sys.c |  |
| 103 | syslog | sys\_syslog | kernel/printk/printk.c |  |
| 104 | getgid | sys\_getgid | kernel/sys.c |  |
| 105 | setuid | sys\_setuid | kernel/sys.c |  |
| 106 | setgid | sys\_setgid | kernel/sys.c |  |
| 107 | geteuid | sys\_geteuid | kernel/sys.c |  |
| 108 | getegid | sys\_getegid | kernel/sys.c |  |
| 109 | setpgid | sys\_setpgid | kernel/sys.c |  |
| 110 | getppid | sys\_getppid | kernel/sys.c |  |
| 111 | getpgrp | sys\_getpgrp | kernel/sys.c |  |
| 112 | setsid | sys\_setsid | kernel/sys.c |  |
| 113 | setreuid | sys\_setreuid | kernel/sys.c |  |
| 114 | setregid | sys\_setregid | kernel/sys.c |  |
| 115 | getgroups | sys\_getgroups | kernel/groups.c |  |
| 116 | setgroups | sys\_setgroups | kernel/groups.c |  |
| 117 | setresuid | sys\_setresuid | kernel/sys.c |  |
| 118 | getresuid | sys\_getresuid | kernel/sys.c |  |
| 119 | setresgid | sys\_setresgid | kernel/sys.c |  |
| 120 | getresgid | sys\_getresgid | kernel/sys.c |  |
| 121 | getpgid | sys\_getpgid | kernel/sys.c |  |
| 122 | setfsuid | sys\_setfsuid | kernel/sys.c |  |
| 123 | setfsgid | sys\_setfsgid | kernel/sys.c |  |
| 124 | getsid | sys\_getsid | kernel/sys.c |  |
| 125 | capget | sys\_capget | kernel/capability.c |  |
| 126 | capset | sys\_capset | kernel/capability.c |  |
| 127 | rt\_sigpending | sys\_rt\_sigpending | kernel/signal.c |  |
| 128 | rt\_sigtimedwait | sys\_rt\_sigtimedwait | kernel/signal.c |  |
| 129 | rt\_sigqueueinfo | sys\_rt\_sigqueueinfo | kernel/signal.c |  |
| 130 | rt\_sigsuspend | sys\_rt\_sigsuspend | kernel/signal.c |  |
| 131 | sigaltstack | sys\_sigaltstack | kernel/signal.c |  |
| 132 | utime | sys\_utime | fs/utimes.c |  |
| 133 | mknod | sys\_mknod | fs/namei.c |  |
| 134 | uselib |  | fs/exec.c |  |
| 135 | personality | sys\_personality | kernel/exec\_domain.c |  |
| 136 | ustat | sys\_ustat | fs/statfs.c |  |
| 137 | statfs | sys\_statfs | fs/statfs.c |  |
| 138 | fstatfs | sys\_fstatfs | fs/statfs.c |  |
| 139 | sysfs | sys\_sysfs | fs/filesystems.c |  |
| 140 | getpriority | sys\_getpriority | kernel/sys.c |  |
| 141 | setpriority | sys\_setpriority | kernel/sys.c |  |
| 142 | sched\_setparam | sys\_sched\_setparam | kernel/sched/core.c |  |
| 143 | sched\_getparam | sys\_sched\_getparam | kernel/sched/core.c |  |
| 144 | sched\_setscheduler | sys\_sched\_setscheduler | kernel/sched/core.c |  |
| 145 | sched\_getscheduler | sys\_sched\_getscheduler | kernel/sched/core.c |  |
| 146 | sched\_get\_priority\_max | sys\_sched\_get\_priority\_max | kernel/sched/core.c |  |
| 147 | sched\_get\_priority\_min | sys\_sched\_get\_priority\_min | kernel/sched/core.c |  |
| 148 | sched\_rr\_get\_interval | sys\_sched\_rr\_get\_interval | kernel/sched/core.c |  |
| 149 | mlock | sys\_mlock | mm/mlock.c |  |
| 150 | munlock | sys\_munlock | mm/mlock.c |  |
| 151 | mlockall | sys\_mlockall | mm/mlock.c |  |
| 152 | munlockall | sys\_munlockall | mm/mlock.c |  |
| 153 | vhangup | sys\_vhangup | fs/open.c |  |
| 154 | modify\_ldt | sys\_modify\_ldt | arch/x86/um/ldt.c |  |
| 155 | pivot\_root | sys\_pivot\_root | fs/namespace.c |  |
| 156 | \_sysctl | sys\_sysctl | kernel/sysctl\_binary.c |  |
| 157 | prctl | sys\_prctl | kernel/sys.c |  |
| 158 | arch\_prctl | sys\_arch\_prctl | arch/x86/um/syscalls\_64.c |  |
| 159 | adjtimex | sys\_adjtimex | kernel/time.c |  |
| 160 | setrlimit | sys\_setrlimit | kernel/sys.c |  |
| 161 | chroot | sys\_chroot | fs/open.c |  |
| 162 | sync | sys\_sync | fs/sync.c |  |
| 163 | acct | sys\_acct | kernel/acct.c |  |
| 164 | settimeofday | sys\_settimeofday | kernel/time.c |  |
| 165 | mount | sys\_mount | fs/namespace.c |  |
| 166 | umount2 | sys\_umount | fs/namespace.c |  |
| 167 | swapon | sys\_swapon | mm/swapfile.c |  |
| 168 | swapoff | sys\_swapoff | mm/swapfile.c |  |
| 169 | reboot | sys\_reboot | kernel/reboot.c |  |
| 170 | sethostname | sys\_sethostname | kernel/sys.c |  |
| 171 | setdomainname | sys\_setdomainname | kernel/sys.c |  |
| 172 | iopl | stub\_iopl | arch/x86/kernel/ioport.c |  |
| 173 | ioperm | sys\_ioperm | arch/x86/kernel/ioport.c |  |
| 174 | create\_module |  | NOT IMPLEMENTED |  |
| 175 | init\_module | sys\_init\_module | kernel/module.c |  |
| 176 | delete\_module | sys\_delete\_module | kernel/module.c |  |
| 177 | get\_kernel\_syms |  | NOT IMPLEMENTED |  |
| 178 | query\_module |  | NOT IMPLEMENTED |  |
| 179 | quotactl | sys\_quotactl | fs/quota/quota.c |  |
| 180 | nfsservctl |  | NOT IMPLEMENTED |  |
| 181 | getpmsg |  | NOT IMPLEMENTED |  |
| 182 | putpmsg |  | NOT IMPLEMENTED |  |
| 183 | afs\_syscall |  | NOT IMPLEMENTED |  |
| 184 | tuxcall |  | NOT IMPLEMENTED |  |
| 185 | security |  | NOT IMPLEMENTED |  |
| 186 | gettid | sys\_gettid | kernel/sys.c |  |
| 187 | readahead | sys\_readahead | mm/readahead.c |  |
| 188 | setxattr | sys\_setxattr | fs/xattr.c |  |
| 189 | lsetxattr | sys\_lsetxattr | fs/xattr.c |  |
| 190 | fsetxattr | sys\_fsetxattr | fs/xattr.c |  |
| 191 | getxattr | sys\_getxattr | fs/xattr.c |  |
| 192 | lgetxattr | sys\_lgetxattr | fs/xattr.c |  |
| 193 | fgetxattr | sys\_fgetxattr | fs/xattr.c |  |
| 194 | listxattr | sys\_listxattr | fs/xattr.c |  |
| 195 | llistxattr | sys\_llistxattr | fs/xattr.c |  |
| 196 | flistxattr | sys\_flistxattr | fs/xattr.c |  |
| 197 | removexattr | sys\_removexattr | fs/xattr.c |  |
| 198 | lremovexattr | sys\_lremovexattr | fs/xattr.c |  |
| 199 | fremovexattr | sys\_fremovexattr | fs/xattr.c |  |
| 200 | tkill | sys\_tkill | kernel/signal.c |  |
| 201 | time | sys\_time | kernel/time.c |  |
| 202 | futex | sys\_futex | kernel/futex.c |  |
| 203 | sched\_setaffinity | sys\_sched\_setaffinity | kernel/sched/core.c |  |
| 204 | sched\_getaffinity | sys\_sched\_getaffinity | kernel/sched/core.c |  |
| 205 | set\_thread\_area |  | arch/x86/kernel/tls.c |  |
| 206 | io\_setup | sys\_io\_setup | fs/aio.c |  |
| 207 | io\_destroy | sys\_io\_destroy | fs/aio.c |  |
| 208 | io\_getevents | sys\_io\_getevents | fs/aio.c |  |
| 209 | io\_submit | sys\_io\_submit | fs/aio.c |  |
| 210 | io\_cancel | sys\_io\_cancel | fs/aio.c |  |
| 211 | get\_thread\_area |  | arch/x86/kernel/tls.c |  |
| 212 | lookup\_dcookie | sys\_lookup\_dcookie | fs/dcookies.c |  |
| 213 | epoll\_create | sys\_epoll\_create | fs/eventpoll.c |  |
| 214 | epoll\_ctl\_old |  | NOT IMPLEMENTED |  |
| 215 | epoll\_wait\_old |  | NOT IMPLEMENTED |  |
| 216 | remap\_file\_pages | sys\_remap\_file\_pages | mm/fremap.c |  |
| 217 | getdents64 | sys\_getdents64 | fs/readdir.c |  |
| 218 | set\_tid\_address | sys\_set\_tid\_address | kernel/fork.c |  |
| 219 | restart\_syscall | sys\_restart\_syscall | kernel/signal.c |  |
| 220 | semtimedop | sys\_semtimedop | ipc/sem.c |  |
| 221 | fadvise64 | sys\_fadvise64 | mm/fadvise.c |  |
| 222 | timer\_create | sys\_timer\_create | kernel/posix-timers.c |  |
| 223 | timer\_settime | sys\_timer\_settime | kernel/posix-timers.c |  |
| 224 | timer\_gettime | sys\_timer\_gettime | kernel/posix-timers.c |  |
| 225 | timer\_getoverrun | sys\_timer\_getoverrun | kernel/posix-timers.c |  |
| 226 | timer\_delete | sys\_timer\_delete | kernel/posix-timers.c |  |
| 227 | clock\_settime | sys\_clock\_settime | kernel/posix-timers.c |  |
| 228 | clock\_gettime | sys\_clock\_gettime | kernel/posix-timers.c |  |
| 229 | clock\_getres | sys\_clock\_getres | kernel/posix-timers.c |  |
| 230 | clock\_nanosleep | sys\_clock\_nanosleep | kernel/posix-timers.c |  |
| 231 | exit\_group | sys\_exit\_group | kernel/exit.c |  |
| 232 | epoll\_wait | sys\_epoll\_wait | fs/eventpoll.c |  |
| 233 | epoll\_ctl | sys\_epoll\_ctl | fs/eventpoll.c |  |
| 234 | tgkill | sys\_tgkill | kernel/signal.c |  |
| 235 | utimes | sys\_utimes | fs/utimes.c |  |
| 236 | vserver |  | NOT IMPLEMENTED |  |
| 237 | mbind | sys\_mbind | mm/mempolicy.c |  |
| 238 | set\_mempolicy | sys\_set\_mempolicy | mm/mempolicy.c |  |
| 239 | get\_mempolicy | sys\_get\_mempolicy | mm/mempolicy.c |  |
| 240 | mq\_open | sys\_mq\_open | ipc/mqueue.c |  |
| 241 | mq\_unlink | sys\_mq\_unlink | ipc/mqueue.c |  |
| 242 | mq\_timedsend | sys\_mq\_timedsend | ipc/mqueue.c |  |
| 243 | mq\_timedreceive | sys\_mq\_timedreceive | ipc/mqueue.c |  |
| 244 | mq\_notify | sys\_mq\_notify | ipc/mqueue.c |  |
| 245 | mq\_getsetattr | sys\_mq\_getsetattr | ipc/mqueue.c |  |
| 246 | kexec\_load | sys\_kexec\_load | kernel/kexec.c |  |
| 247 | waitid | sys\_waitid | kernel/exit.c |  |
| 248 | add\_key | sys\_add\_key | security/keys/keyctl.c |  |
| 249 | request\_key | sys\_request\_key | security/keys/keyctl.c |  |
| 250 | keyctl | sys\_keyctl | security/keys/keyctl.c |  |
| 251 | ioprio\_set | sys\_ioprio\_set | fs/ioprio.c |  |
| 252 | ioprio\_get | sys\_ioprio\_get | fs/ioprio.c |  |
| 253 | inotify\_init | sys\_inotify\_init | fs/notify/inotify/inotify\_user.c |  |
| 254 | inotify\_add\_watch | sys\_inotify\_add\_watch | fs/notify/inotify/inotify\_user.c |  |
| 255 | inotify\_rm\_watch | sys\_inotify\_rm\_watch | fs/notify/inotify/inotify\_user.c |  |
| 256 | migrate\_pages | sys\_migrate\_pages | mm/mempolicy.c |  |
| 257 | openat | sys\_openat | fs/open.c |  |
| 258 | mkdirat | sys\_mkdirat | fs/namei.c |  |
| 259 | mknodat | sys\_mknodat | fs/namei.c |  |
| 260 | fchownat | sys\_fchownat | fs/open.c |  |
| 261 | futimesat | sys\_futimesat | fs/utimes.c |  |
| 262 | newfstatat | sys\_newfstatat | fs/stat.c |  |
| 263 | unlinkat | sys\_unlinkat | fs/namei.c |  |
| 264 | renameat | sys\_renameat | fs/namei.c |  |
| 265 | linkat | sys\_linkat | fs/namei.c |  |
| 266 | symlinkat | sys\_symlinkat | fs/namei.c |  |
| 267 | readlinkat | sys\_readlinkat | fs/stat.c |  |
| 268 | fchmodat | sys\_fchmodat | fs/open.c |  |
| 269 | faccessat | sys\_faccessat | fs/open.c |  |
| 270 | pselect6 | sys\_pselect6 | fs/select.c |  |
| 271 | ppoll | sys\_ppoll | fs/select.c |  |
| 272 | unshare | sys\_unshare | kernel/fork.c |  |
| 273 | set\_robust\_list | sys\_set\_robust\_list | kernel/futex.c |  |
| 274 | get\_robust\_list | sys\_get\_robust\_list | kernel/futex.c |  |
| 275 | splice | sys\_splice | fs/splice.c |  |
| 276 | tee | sys\_tee | fs/splice.c |  |
| 277 | sync\_file\_range | sys\_sync\_file\_range | fs/sync.c |  |
| 278 | vmsplice | sys\_vmsplice | fs/splice.c |  |
| 279 | move\_pages | sys\_move\_pages | mm/migrate.c |  |
| 280 | utimensat | sys\_utimensat | fs/utimes.c |  |
| 281 | epoll\_pwait | sys\_epoll\_pwait | fs/eventpoll.c |  |
| 282 | signalfd | sys\_signalfd | fs/signalfd.c |  |
| 283 | timerfd\_create | sys\_timerfd\_create | fs/timerfd.c |  |
| 284 | eventfd | sys\_eventfd | fs/eventfd.c |  |
| 285 | fallocate | sys\_fallocate | fs/open.c |  |
| 286 | timerfd\_settime | sys\_timerfd\_settime | fs/timerfd.c |  |
| 287 | timerfd\_gettime | sys\_timerfd\_gettime | fs/timerfd.c |  |
| 288 | accept4 | sys\_accept4 | net/socket.c |  |
| 289 | signalfd4 | sys\_signalfd4 | fs/signalfd.c |  |
| 290 | eventfd2 | sys\_eventfd2 | fs/eventfd.c |  |
| 291 | epoll\_create1 | sys\_epoll\_create1 | fs/eventpoll.c |  |
| 292 | dup3 | sys\_dup3 | fs/file.c |  |
| 293 | pipe2 | sys\_pipe2 | fs/pipe.c |  |
| 294 | inotify\_init1 | sys\_inotify\_init1 | fs/notify/inotify/inotify\_user.c |  |
| 295 | preadv | sys\_preadv | fs/read\_write.c |  |
| 296 | pwritev | sys\_pwritev | fs/read\_write.c |  |
| 297 | rt\_tgsigqueueinfo | sys\_rt\_tgsigqueueinfo | kernel/signal.c |  |
| 298 | perf\_event\_open | sys\_perf\_event\_open | kernel/events/core.c |  |
| 299 | recvmmsg | sys\_recvmmsg | net/socket.c |  |
| 300 | fanotify\_init | sys\_fanotify\_init | fs/notify/fanotify/fanotify\_user.c |  |
| 301 | fanotify\_mark | sys\_fanotify\_mark | fs/notify/fanotify/fanotify\_user.c |  |
| 302 | prlimit64 | sys\_prlimit64 | kernel/sys.c |  |
| 303 | name\_to\_handle\_at | sys\_name\_to\_handle\_at | fs/fhandle.c |  |
| 304 | open\_by\_handle\_at | sys\_open\_by\_handle\_at | fs/fhandle.c |  |
| 305 | clock\_adjtime | sys\_clock\_adjtime | kernel/posix-timers.c |  |
| 306 | syncfs | sys\_syncfs | fs/sync.c |  |
| 307 | sendmmsg | sys\_sendmmsg | net/socket.c |  |
| 308 | setns | sys\_setns | kernel/nsproxy.c |  |
| 309 | getcpu | sys\_getcpu | kernel/sys.c |  |
| 310 | process\_vm\_readv | sys\_process\_vm\_readv | mm/process\_vm\_access.c |  |
| 311 | process\_vm\_writev | sys\_process\_vm\_writev | mm/process\_vm\_access.c |  |
| 312 | kcmp | sys\_kcmp | kernel/kcmp.c |  |
| 313 | finit\_module | sys\_finit\_module | kernel/module.c |  |

### 4.4.1 虚拟文件系统

https://blog.csdn.net/vertor11/article/details/79622694

#### 4.4.4.1 proc文件系统 以后再看

<https://www.ibm.com/developerworks/cn/linux/l-proc.html>

代码:

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/init.h>

#include <linux/version.h>

#include <linux/proc\_fs.h>

#include <linux/seq\_file.h>

static unsigned int variable;

static struct proc\_dir\_entry \*test\_dir, \*test\_entry;

#if LINUX\_VERSION\_CODE < KERNEL\_VERSION(3, 10, 0)

//other code

#else

static int test\_proc\_show(struct seq\_file \*seq, void \*v)

{

    unsigned int \*ptr\_var = seq->private;

    seq\_printf(seq, "%u\n", \*ptr\_var);

    return 0;

}

static ssize\_t test\_proc\_write(struct file \*file, const char \_\_user \*buffer,

    size\_t count, loff\_t \*ppos)

{

    struct seq\_file \*seq = file->private\_data;

    unsigned int \*ptr\_var = seq->private;

    \*ptr\_var = simple\_strtoul(buffer, NULL, 10);

    return count;

}

static int test\_proc\_open(struct inode \*inode, struct file \*file)

{

    return single\_open(file, test\_proc\_show, PDE\_DATA(inode));

}

static const struct file\_operations test\_proc\_fops =

{

    .owner = THIS\_MODULE,

    .open = test\_proc\_open,

    .read = seq\_read,

    .write = test\_proc\_write,

    .llseek = seq\_lseek,

    .release = single\_release,

};

#endif

static \_\_init int test\_proc\_init(void)

{

    test\_dir = proc\_mkdir("test\_dir", NULL);//在proc创建一个test\_dir文件夹

    if (test\_dir) {

        #if LINUX\_VERSION\_CODE < KERNEL\_VERSION(3, 10, 0)

            //other code

        #else

//test\_rw 文件

//0666 文件权限

//test\_dir 在那个文件夹下面创建文件

// test\_proc\_fops 文件操作方法回调

            test\_entry = proc\_create\_data("test\_rw", 0666, test\_dir, &test\_proc\_fops, &variable);

            if (test\_entry) {

                return 0;

            }

        #endif

    }

    return -ENOMEM;

}

module\_init(test\_proc\_init);

static \_\_exit void test\_proc\_cleanup(void)

{

    remove\_proc\_entry("test\_rw", test\_dir);

    remove\_proc\_entry("test\_dir", NULL);

}

module\_exit(test\_proc\_cleanup);

MODULE\_AUTHOR("Barry Song <baohua@kernel.org>");

MODULE\_DESCRIPTION("proc example");

MODULE\_LICENSE("GPL v2");

#### 4.4.4.1 sysfs文件系统

#### 4.4.4.1 debugfs文件系统

### 4.4.1 ioctl接口

### 4.4.1 netlink

## 4.1页面机制

# 第六章linux子系统总结

## 6.1 输入子系统

### 6.6.1三个重要的结构体

综述输入子系统框图：



Input.c 1 创建一个bus/input文件夹

2 创建类 /sys/class/ （类注册）

3 输入类设备号是13（[root@192 input]# cat /proc/devices）

#### 4.4.4.1 struct input\_dev

试验代码：/LinuxTestCode/module/input\_subsystem\_button

Struct input\_dev（代表输入设备）

Include/linux/input.h

struct [**input\_dev**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_dev) {

const char \*name;

const char \*[**phys**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/phys);

const char \*[**uniq**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/uniq);

struct [**input\_id**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_id) id;//设备用户信息

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/long) propbit[[**BITS\_TO\_LONGS**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/BITS_TO_LONGS)([**INPUT\_PROP\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/INPUT_PROP_CNT))];

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/long) evbit[[**BITS\_TO\_LONGS**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/BITS_TO_LONGS)([**EV\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/EV_CNT))];

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/long) keybit[[**BITS\_TO\_LONGS**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/BITS_TO_LONGS)([**KEY\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/KEY_CNT))];

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/long) relbit[[**BITS\_TO\_LONGS**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/BITS_TO_LONGS)([**REL\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/REL_CNT))];

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/long) absbit[[**BITS\_TO\_LONGS**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/BITS_TO_LONGS)([**ABS\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/ABS_CNT))];

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/long) mscbit[[**BITS\_TO\_LONGS**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/BITS_TO_LONGS)([**MSC\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/MSC_CNT))];

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/long) ledbit[[**BITS\_TO\_LONGS**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/BITS_TO_LONGS)([**LED\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/LED_CNT))];

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/long) sndbit[[**BITS\_TO\_LONGS**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/BITS_TO_LONGS)([**SND\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/SND_CNT))];

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/long) ffbit[[**BITS\_TO\_LONGS**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/BITS_TO_LONGS)([**FF\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/FF_CNT))];

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/long) swbit[[**BITS\_TO\_LONGS**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/BITS_TO_LONGS)([**SW\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/SW_CNT))];

unsigned int hint\_events\_per\_packet;

unsigned int keycodemax;

unsigned int keycodesize;

void \*keycode;

int (\*[**setkeycode**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/setkeycode))(struct [**input\_dev**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_dev) \*dev,

const struct [**input\_keymap\_entry**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_keymap_entry) \*ke,

unsigned int \*old\_keycode);

int (\*[**getkeycode**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/getkeycode))(struct [**input\_dev**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_dev) \*dev,

struct [**input\_keymap\_entry**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_keymap_entry) \*ke);

struct [**ff\_device**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/ff_device) \*[**ff**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/ff);

unsigned int repeat\_key;

struct [**timer\_list**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/timer_list) [**timer**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/timer);

int [**rep**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/rep)[[**REP\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/REP_CNT)];

struct [**input\_mt**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_mt) \*[**mt**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/mt);

struct [**input\_absinfo**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_absinfo) \*absinfo;

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/long) [**key**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/key)[[**BITS\_TO\_LONGS**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/BITS_TO_LONGS)([**KEY\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/KEY_CNT))];

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/long) [**led**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/led)[[**BITS\_TO\_LONGS**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/BITS_TO_LONGS)([**LED\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/LED_CNT))];

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/long) snd[[**BITS\_TO\_LONGS**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/BITS_TO_LONGS)([**SND\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/SND_CNT))];

unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/long) [**sw**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/sw)[[**BITS\_TO\_LONGS**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/BITS_TO_LONGS)([**SW\_CNT**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/SW_CNT))];

int (\*[**open**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/open))(struct [**input\_dev**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_dev) \*dev);

void (\*[**close**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/close))(struct [**input\_dev**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_dev) \*dev);

int (\*[**flush**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/flush))(struct [**input\_dev**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_dev) \*dev, struct [**file**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/file) \*[**file**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/file));

int (\*event)(struct [**input\_dev**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_dev) \*dev, unsigned int type, unsigned int [**code**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/code), int value);

struct [**input\_handle**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_handle) [**\_\_rcu**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/__rcu) \*grab;

[**spinlock\_t**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/spinlock_t) event\_lock;

struct [**mutex**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/mutex) [**mutex**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/mutex);

unsigned int [**users**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/users);

[**bool**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/bool) going\_away;

struct [**device**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/device) dev;

struct [**list\_head**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/list_head) h\_list;

struct [**list\_head**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/list_head) [**node**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/node);

unsigned int [**num\_vals**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/num_vals);

unsigned int [**max\_vals**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/max_vals);

struct [**input\_value**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_value) \*vals;

[**bool**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/bool) devres\_managed;

};

#### 4.4.4.1 struct input\_handler

Struct input\_handler(输入设备的处理方法)

struct [**input\_handler**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_handler) {

void \*[**private**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/private);

void (\*event)(struct [**input\_handle**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_handle) \*[**handle**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/handle), unsigned int type, unsigned int [**code**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/code), int value);

void (\*[**events**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/events))(struct [**input\_handle**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_handle) \*[**handle**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/handle),

const struct [**input\_value**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_value) \*vals, unsigned int count);

[**bool**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/bool) (\*[**filter**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/filter))(struct [**input\_handle**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_handle) \*[**handle**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/handle), unsigned int type, unsigned int [**code**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/code), int value);

[**bool**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/bool) (\*[**match**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/match))(struct [**input\_handler**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_handler) \*[**handler**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/handler), struct [**input\_dev**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_dev) \*dev);

int (\*[**connect**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/connect))(struct [**input\_handler**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_handler) \*[**handler**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/handler), struct [**input\_dev**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_dev) \*dev, const struct [**input\_device\_id**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_device_id) \*id);

void (\*[**disconnect**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/disconnect))(struct [**input\_handle**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_handle) \*[**handle**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/handle));

void (\*start)(struct [**input\_handle**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_handle) \*[**handle**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/handle));

[**bool**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/bool) legacy\_minors;

int [**minor**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/minor);

const char \*name;

const struct [**input\_device\_id**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/input_device_id) \*[**id\_table**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/id_table);

struct [**list\_head**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/list_head) h\_list;

struct [**list\_head**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/list_head) [**node**](https://elixir.bootlin.com/linux/v4.9.35/C/ident/node);

};

Struct input\_handle（关联input\_dev和input\_handler）

#### 4.4.4.1 输入子系统

## 6.2 中断子系统

### 6.2.1中断子系统结构体

# 第六章linux文件系统

## 6.1 file struct结构体

Include/linux/fs.h

struct [**file**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/file) {

union {

struct [**llist\_node**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/llist_node) [**fu\_llist**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/fu_llist);

struct [**rcu\_head**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/rcu_head) [**fu\_rcuhead**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/fu_rcuhead);

} [**f\_u**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_u);

struct [**path**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/path) [**f\_path**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_path);

struct [**inode**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/inode) \*[**f\_inode**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_inode); */\* cached value \*/*

const struct [**file\_operations**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/file_operations) \*[**f\_op**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_op);

*/\**

*\* Protects f\_ep\_links, f\_flags.*

*\* Must not be taken from IRQ context.*

*\*/*

[**spinlock\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/spinlock_t) [**f\_lock**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_lock);

[**atomic\_long\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/atomic_long_t) [**f\_count**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_count);

unsigned int [**f\_flags**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_flags);

[**fmode\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/fmode_t) [**f\_mode**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_mode);

struct [**mutex**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/mutex) [**f\_pos\_lock**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_pos_lock);

[**loff\_t**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/loff_t) [**f\_pos**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_pos);

struct [**fown\_struct**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/fown_struct) [**f\_owner**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_owner);

const struct [**cred**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/cred) \*[**f\_cred**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_cred);

struct [**file\_ra\_state**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/file_ra_state) [**f\_ra**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_ra);

[**u64**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/u64) [**f\_version**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_version);

#ifdef [**CONFIG\_SECURITY**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_SECURITY)

void \*[**f\_security**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_security);

#endif

*/\* needed for tty driver, and maybe others \*/*

void \*[**private\_data**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/private_data);

#ifdef [**CONFIG\_EPOLL**](https://elixir.bootlin.com/linux/v4.9.39/K/ident/CONFIG_EPOLL)

*/\* Used by fs/eventpoll.c to link all the hooks to this file \*/*

struct [**list\_head**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/list_head) [**f\_ep\_links**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_ep_links);

struct [**list\_head**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/list_head) [**f\_tfile\_llink**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_tfile_llink);

#endif */\* #ifdef CONFIG\_EPOLL \*/*

struct [**address\_space**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/address_space) \*[**f\_mapping**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/f_mapping);

} [**\_\_attribute\_\_**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/__attribute__)(([**aligned**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/aligned)(4))); */\* lest something weird decides that 2 is OK \*/*

# 第六章重要命令简介

## 5.1 nm命令

# 第七章面试题汇总（linux内核内存管理总结）

<https://mp.weixin.qq.com/s/TO8mzZO18_NBEthPq7Y8Og>

## 6.1 nm命令

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A Global absolute 符号。 a Local absolute 符号。 B Global bss 符号。 b Local bss 符号。 D Global data 符号。 d Local data 符号。 f 源文件名称符号。 T Global text 符号。 t Local text 符号。 U 未定义符号。 | | |  |  |  |
| 0000000000601048 | b | a3.3406 |  |  |  |
| 000000000060104c | B | a5 |  |  |  |
| 0000000000601040 | D | a6 |  |  |  |
| 0000000000601044 | B | \_\_bss\_start |  |  |  |
| 0000000000601044 | b | completed.7594 |  |  |  |
| 0000000000601030 | D | \_\_data\_start |  |  |  |
| 0000000000601030 | W | data\_start |  |  |  |
| 00000000004004a0 | t | deregister\_tm\_clones |  |  |  |
| 0000000000400520 | t | \_\_do\_global\_dtors\_aux |  |  |  |
| 0000000000600e18 | t | \_\_do\_global\_dtors\_aux\_fini\_array\_entry |  |  |  |
| 0000000000601038 | D | \_\_dso\_handle |  | Stack |  |
| 0000000000600e28 | d | \_DYNAMIC |  |  |  |
| 0000000000601044 | D | \_edata |  |  |  |
| 0000000000601050 | B | \_end |  |  |  |
| 0000000000400624 | T | \_fini |  |  |  |
| 0000000000400540 | t | frame\_dummy |  |  |  |
| 0000000000600e10 | t | \_\_frame\_dummy\_init\_array\_entry |  | mmap |  |
| 0000000000400780 | r | \_\_FRAME\_END\_\_ |  |  |  |
| 0000000000400566 | T | function |  |  |  |
| 0000000000601000 | d | \_GLOBAL\_OFFSET\_TABLE\_ |  |  |  |
|  |  | w \_\_gmon\_start\_\_ |  |  |  |
| 0000000000400634 | r | \_\_GNU\_EH\_FRAME\_HDR |  | Heap |  |
| 0000000000400400 | T | \_init |  |  | 未初始化变量 |
| 0000000000600e18 | t | \_\_init\_array\_end |  |  |  |
| 0000000000600e10 | t | \_\_init\_array\_start |  | Bss segment |  |
| 0000000000400630 | R | \_IO\_stdin\_used |  |  |  |
|  |  | w \_ITM\_deregisterTMCloneTable |  |  |  |
|  |  | w \_ITM\_registerTMCloneTable |  | Data segment |  |
| 0000000000600e20 | d | \_\_JCR\_END\_\_ |  |  |  |
| 0000000000600e20 | d | \_\_JCR\_LIST\_\_ |  |  |  |
|  |  | w \_Jv\_RegisterClasses |  | Text segment |  |
| 0000000000400620 | T | \_\_libc\_csu\_fini |  |  |  |
| 00000000004005b0 | T | \_\_libc\_csu\_init |  |  |  |
|  |  | U \_\_libc\_start\_main@@GLIBC\_2.2.5 |  |  |  |
| 0000000000400571 | T | main |  |  |  |
|  |  | U malloc@@GLIBC\_2.2.5 |  |  |  |
| 00000000004004e0 | t | register\_tm\_clones |  |  |  |
|  |  | U sleep@@GLIBC\_2.2.5 |  |  |  |
| 0000000000400470 | T | \_start |  |  |  |
| 0000000000601048 | D | \_\_TMC\_END\_\_ |  |  |  |

## 6.1在系统启动时，ARM Linux内核如何知道系统中有多大的内存空间？

通过dts可以查找到

1. 在32bit Linux内核中，用户空间和内核空间的比例通常是3:1，可以修改成2:2吗？

可以

1. 物理内存页面如何添加到伙伴系统中，是一页一页添加，还是以2的几次幂来加入呢？

2的幂次方添加

1. 内核的一级页表存放在什么地方？二级页表又存放在什么地方？

32位：

64位：

1. 用户进程的一级页表存放在什么地方？二级页表呢？

32位:

64位：

1. 在ARM32系统中，页表是如何映射的？在ARM64系统中，页表又是如何映射的？
2. 请简述Linux内核在理想情况下页面分配器(page allocator)是如何分配出连续物理页面的。
3. 在页面分配器中，如何从分配掩码(gfp\_mask)中确定可以从哪些zone中分配内存？
4. 页面分配器是按照什么方向来扫描zone的？
5. 为用户进程分配物理内存，分配掩码应该选用GFP\_KERNEL，还是GFP\_HIGHUSER\_MOVABLE呢？

Alloc\_pages(gfp\_t mask,int num)->最终会去调用\_\_get\_free\_page()

Mask= gfp\_kernel 是内核正常分配内存的方式，无内存引起休眠

Mssk = gfp\_highuser\_movable

Mask = gfp\_user 用户进程分配内存

详情见/root/module/gfp\_kernel\_user/gfp\_kernel\_user.c

1. slab分配器是如何分配和释放小块内存的？
2. slab分配器中有一个着色的概念(cache color)，着色有什么作用？
3. slab分配其中的slab对象有没有根据Per-CPU做一些优化？
4. slab增长并导致大量不用的空闲对象，该如何解决？

关于slab内容详解：<https://blog.csdn.net/lukuen/article/details/6935068>

1. 请问kmalloc、vmalloc和malloc之间有什么区别以及实现上的差异？

Malloc 属于用户空间申请的内存在堆上的申请

Vmalloc 内核空间申请

Kmalloc 内核申请空间

1. 使用用户态的API函数malloc()分配内存时，会马上为其分配物理内存吗？

不会，当用到会产生缺页中断，分配屋里内存

疑问：如果不断的分配

While(1)

{

P = malloc(10);//这样会造成严重的物理内存泄漏

}

1. 假设不考虑libc的因素，malloc分配100Byte，那么实际上内核是为其分配100Byte吗？

大于100byte内存

例子：root/code/user/malloc/malloc.c

注释：如下是libc的原理

cat 23084/maps

heap一次分配132K内存

013eb000-0140c000 rw-p 00000000 00:00 0 [heap]

小内存的分配是使用slab分配：

1. 假设两个用户进程打印的malloc()分配的虚拟地址是一样的，那么在内核中这两块虚拟内存是否打架了呢？

两个相同的虚拟内存地址物理地址不一样：

1. vm\_normal\_page()函数返回的是什么样页面的struct page数据结构？为什么内存管理代码中需要这个函数？brk中涉及内存管理中的几个重要函数

1.get\_user\_pages()

2.follow\_page()

3.vm\_normal\_page()

1. 请简述get\_user\_page()函数的作用和实现流程？
2. 请简述follow\_page()函数的作用和实现流程？
3. 请简述私有映射和共享映射的区别。

https://blog.csdn.net/qq\_19004627/article/details/79255054

多个进程对同一块内存操作，根据属性的不同分为：

参数prot：映射区域的保护方式。可以为以下几种方式的组合：

PROT\_EXEC 映射区域可被执行

PROT\_READ 映射区域可被读取

PROT\_WRITE 映射区域可被写入

PROT\_NONE 映射区域不能存取

参数flags：影响映射区域的各种特性。在调用mmap()时必须要指定MAP\_SHARED 或MAP\_PRIVATE。

MAP\_FIXED 如果参数start所指的地址无法成功建立映射时，则放弃映射，不对地址做修正。通常不鼓励用此旗标。

MAP\_SHARED对映射区域的写入数据会复制回文件内，而且允许其他映射该文件的进程共享。

MAP\_PRIVATE 对映射区域的写入操作会产生一个映射文件的复制，即私人的“写入时复制”（copy on write）对此区域作的任何修改都不会写回原来的文件内容。

MAP\_ANONYMOUS建立匿名映射。此时会忽略参数fd，不涉及文件，而且映射区域无法和其他进程共享。

MAP\_DENYWRITE只允许对映射区域的写入操作，其他对文件直接写入的操作将会被拒绝。

MAP\_LOCKED 将映射区域锁定住，这表示该区域不会被置换（swap）

私有映射：

(mapped = (char \*)mmap(NULL, st\_size, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE, fd, 0))

共享映射：

(mapped = (char \*)mmap(NULL, st\_size, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0))

1. 为什么第二次调用mmap时，Linux内核没有捕捉到地址重叠并返回失败呢？
2. struct page数据结构中的\_count和\_mapcount有什么区别？

\_\_count成员

\_count表示内核中引用该页面的次数。

在memmap\_init\_zone函数把set\_page\_count(page,0)把pagecount设置为-1，然后在free\_all\_bootmem\_core调用set\_page\_refs(page, 0);在set\_page\_refs中set\_page\_count(page,1)把pagecount设置为0，这样初始化buddy的页的pagecount为0。

\_\_count=0: 表示页是空闲的

————————————————

注意区分\_count和\_mapcount，\_mapcount表示的是映射次数，而\_count表示的是使用次数；被映射了不一定在使用，但要使用必须先映射。

原文链接：https://blog.csdn.net/chengbeng1745/article/details/86598604

总结page结构体

1. 匿名页面和page cache页面有什么区别？
2. struct page数据结构中有一个锁，请问trylock\_page()和lock\_page()有什么区别？
3. 在Linux 2.4.x内核中，如何从一个page找到所有映射该页面的VMA？反响映射可以带来哪些便利？

由虚拟地址—->物理地址（正向映射）

物理地址—>虚拟地址（反向映射）

1. 阅读Linux 4.0内核RMAP机制的代码，画出父子进程之间VMA、AVC、anon\_vma和page等数据结构之间的关系图。
2. 在Linux 2.6.34中，RMAP机制采用了新的实现，在Linux 2.6.33和之前的版本中称为旧版本RMAP机制。那么在旧版本RMAP机制中，如果父进程有1000个子进程，每个子进程都有一个VMA，这个VMA里面有1000个匿名页面，当所有的子进程的VMA同时发生写复制时会是什么情况呢？  
   当page加入lru链表中，被其他线程释放了这个page，那么lru链表如何知道这个page已经被释放了。
3. kswapd内核线程何时会被唤醒？
4. LRU链表如何知道page的活动频繁程度？
5. kswapd按照什么原则来换出页面？
6. kswapd按照什么方向来扫描zone？
7. kswapd以什么标准来退出扫描LRU？
8. 手持设备例如Android系统，没有swap分区或者swap文件，kswapd会扫描匿名页面LRU吗？
9. swappiness的含义是什么？kswapd如何计算匿名页面和page cache之间的扫描比重？
10. 当系统充斥着大量只访问一次的文件访问(use-one streaming IO)时，kswapd如何来规避这种风暴？
11. 在回收page cache时，对于dirty的page cache，kswapd会马上回写吗？
12. 内核有哪些页面会被kswapd写回交换分区？
13. ARM32 Linux如何模拟这个Linux版本的L\_PTE\_YOUNG比特位呢？
14. 如何理解Refault Distance算法？
15. 请简述匿名页面的生命周期。在什么情况下会产生匿名页面？在什么条件下会释放匿名页面？
16. KSM是基于什么原理来合并页面的？
17. 在KSM机制里，合并过程中把page设置成写保护的函数write\_protect\_page()有这样一个判断：这个判断的依据是什么？
18. 如果多个VMA的虚拟页面同时映射了同一个匿名页面，那么此时page->index应该等于多少？
19. 为什么Dirty COW小程序可以修改一个只读文件的内容？
20. 在Dirty COW内存漏洞中，如果Diryt COW程序没有madviseThread线程，即只有procselfmemThread线程，能否修改foo文件的内容呢？
21. 假设在内核空间获取了某个文件对应的page cache页面的struct page数据结构，而对应的VMA属性是只读，那么内核空间是否可以成功修改该文件呢？
22. 如果用户进程使用只读属性(PROT\_READ)来mmap映射一个文件到用户空间，然后使用memcpy来写这段内存空间，会是什么样的情况？
23. 请画出内存管理中常用的数据结构的关系图，如mm\_struct、vma、vaddr、page、pfn、pte、zone、paddr和pg\_data等，并思考如下转换关系。
24. 请画出在最糟糕的情况下分配若干个连续物理页面的流程图。
25. 在Android中新添加了LMK(Low Memory Killer)，请描述LMK和OOM Killer之间的关系。
26. 请描述一致性DMA映射dma\_alloc\_coherent()函数在AEM中是如何管理cache一致性的？
27. 请描述流式DMA映射dma\_map\_single()函数在ARM中是如何管理cache一致性的？
28. 为什么在Linux 4.8内核中要把基于zone的LRU链表机制迁移到基于Node呢？

# 第7章 杂乱的命令总结

## 7.1查看设备树，存放在系统的位置

**a. /sys/firmware/fdt**

进入/sys/firmware目录后便可看到二个文件，一个是devicetree文件夹，另一个是fdt（原始dtb文件，可以用hexdump -C fdt 将其打印出来查看就会发现里面的数据和dtb文件是一致的）。

**b. /sys/firmware/devicetree**

以目录结构呈现的dtb文件。 根节点对应base目录, 每一个节点对应一个目录, 每一个属性对应一个文件

**c. /sys/devices/platform**

系统中所有的platform\_device, 有来自设备树的, 也有来有.c文件中注册的  
对于来自设备树的platform\_device,可以进入 /sys/devices/platform/<设备名>/of\_node 查看它的设备树属性（例如进入/sys/devices/platform/led/后若发现该目录下有of\_node节点，就表明该platform\_device来自设备树）

**d. /proc/device-tree**

是链接文件, 指向 /sys/firmware/devicetree/base

## 7.2 gdb使用技巧

|  |  |  |
| --- | --- | --- |
| 命令 | 命令缩写 | 命令说明 |
| list | l | 显示多行源代码 |
| break | b | 设置断点,程序运行到断点的位置会停下来 |
| info | i | 描述程序的状态 |
| run | r | 开始运行程序 |
| display | disp | 跟踪查看某个变量,每次停下来都显示它的值 |
| step | s | 执行下一条语句,如果该语句为函数调用,则进入函数执行其中的第一条语句 |
| next | n | 执行下一条语句,如果该语句为函数调用,不会进入函数内部执行(即不会一步步地调试函数内部语句) |
| print | p | 打印内部变量值 |
| continue | c | 继续程序的运行,直到遇到下一个断点 |
| set var name=v |  | 设置变量的值 |
| start | st | 开始执行程序,在main函数的第一条语句前面停下来 |
| file |  | 装入需要调试的程序 |
| kill | k | 终止正在调试的程序 |
| watch |  | 监视变量值的变化 |
| backtrace | bt | 产看函数调用信息(堆栈) |
| frame | f | 查看栈帧 |
| quit | q | 退出GDB环境 |
| disassemble |  | 反汇编指令 |
|  | X | 按照16进制显示变量 |

## 7.3 pmap使用技巧



## 7.4 valgrind内存泄漏检测工具使用

引用地址：https://blog.csdn.net/sunmenggmail/article/details/10543483



编译源代码加入-g显示源码（gcc bad\_memory.c -g）

将程序编译生成可执行文件后执行：valgrind --leak-check=full ./程序名

输出结果如下：

==4832== Memcheck, a memory error detector  
==4832== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.  
==4832== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info  
==4832== Command: ./tmp  
==4832==   
==4832== Invalid write of size 4      // 内存越界  
==4832==    at 0x804843F: test (in /home/yanghao/Desktop/testC/testmem/tmp)  
==4832==    by 0x804848D: main (in /home/yanghao/Desktop/testC/testmem/tmp)  
==4832==  Address 0x41a6050 is 0 bytes after a block of size 40 alloc'd  
==4832==    at 0x4026864: malloc (vg\_replace\_malloc.c:236)  
==4832==    by 0x8048435: test (in /home/yanghao/Desktop/testC/testmem/tmp)  
==4832==    by 0x804848D: main (in /home/yanghao/Desktop/testC/testmem/tmp)  
==4832==   
==4832== Source and destination overlap in memcpy(0x41a602c, 0x41a6028, 5) // 踩内存  
==4832==    at 0x4027BD6: memcpy (mc\_replace\_strmem.c:635)  
==4832==    by 0x8048461: test (in /home/yanghao/Desktop/testC/testmem/tmp)  
==4832==    by 0x804848D: main (in /home/yanghao/Desktop/testC/testmem/tmp)  
==4832==   
==4832== Invalid free() / delete / delete[] // 重复释放  
==4832==    at 0x4025BF0: free (vg\_replace\_malloc.c:366)  
==4832==    by 0x8048477: test (in /home/yanghao/Desktop/testC/testmem/tmp)  
==4832==    by 0x804848D: main (in /home/yanghao/Desktop/testC/testmem/tmp)  
==4832==  Address 0x41a6028 is 0 bytes inside a block of size 40 free'd  
==4832==    at 0x4025BF0: free (vg\_replace\_malloc.c:366)  
==4832==    by 0x804846C: test (in /home/yanghao/Desktop/testC/testmem/tmp)  
==4832==    by 0x804848D: main (in /home/yanghao/Desktop/testC/testmem/tmp)  
==4832==   
==4832== Use of uninitialised value of size 4 // 非法指针  
==4832==    at 0x804847B: test (in /home/yanghao/Desktop/testC/testmem/tmp)  
==4832==    by 0x804848D: main (in /home/yanghao/Desktop/testC/testmem/tmp)  
==4832==   
==4832==   
==4832== Process terminating with default action of signal 11 (SIGSEGV) //由于非法指针赋值导致的程序崩溃  
==4832==  Bad permissions for mapped region at address 0x419FFF4  
==4832==    at 0x804847B: test (in /home/yanghao/Desktop/testC/testmem/tmp)  
==4832==    by 0x804848D: main (in /home/yanghao/Desktop/testC/testmem/tmp)  
==4832==   
==4832== HEAP SUMMARY:  
==4832==     in use at exit: 0 bytes in 0 blocks  
==4832==   total heap usage: 1 allocs, 2 frees, 40 bytes allocated  
==4832==   
==4832== All heap blocks were freed -- no leaks are possible  
==4832==   
==4832== For counts of detected and suppressed errors, rerun with: -v  
==4832== Use --track-origins=yes to see where uninitialised values come from  
==4832== ERROR SUMMARY: 4 errors from 4 contexts (suppressed: 11 from 6)  
Segmentation fault

从valgrind的检测输出结果看，这几个错误都找了出来。

## 7.5 使用内核打印用户进程的和kernel进程的所有信息

代码见LinuxTestCode/module/thread\_info

## 7.6 用户进程和内核进程交换数据的方式

/LinuxTestCode/user/use\_c\_kernel

## 7.7 vmalloc\_to\_page\_pfn

代码见：/LinuxTestCode/module/08\_map\_driver

Remap\_pfn\_range() 需要连续的空间(建立也表项)

Fault() 分配非连续的空间（建立也表项）

Vmalloc\_to\_page()

Vmalloc\_to\_pfn()

测试步骤：

1 由Vmalloc\_to\_page() 求出page的地址



2 打印此地址结构体参数crash> struct page ffffef5c40e24b00



3 打印所有的page地址一个pagesize(64=0x40)

代码：/LinuxTestCode/module/10\_read\_memory\_info



## 7.8 内核监控负载均衡

代码：/LinuxTestCode/module/12\_load\_monitor

Uptime

Nproc

模拟高负载的状态

while :;do :;done &

cat /proc/loadavg  可以看到当前系统的load

[root@localhost ~]# cat /proc/loadavg

0.02 0.01 0.00 1/349 87344

前面三个值分别对应系统当前1分钟、5分钟、15分钟内的平均load

## 7.9 获取内核函数地址的几种方法

1 使用System.map

[root@localhost linux-4.9.39]# cat System.map | grep avenrun

ffffffff810c1810 T get\_avenrun

ffffffff81b04400 r \_\_ksymtab\_avenrun

ffffffff81b237b8 r \_\_kcrctab\_avenrun

ffffffff81b3523a r \_\_kstrtab\_avenrun

ffffffff82003750 B avenrun

2 使用vmlinux

[root@localhost linux-4.9.39]# nm vmlinux | grep avenrun

ffffffff82003750 B avenrun

00000000f1e98c74 A \_\_crc\_avenrun

ffffffff810c1810 T get\_avenrun

ffffffff81b237b8 r \_\_kcrctab\_avenrun

ffffffff81b3523a r \_\_kstrtab\_avenrun

ffffffff81b04400 r \_\_ksymtab\_avenrun

3

[root@localhost linux-4.9.39]# cat /proc/kallsyms | grep avenrun

ffffffffb68c1810 T get\_avenrun

ffffffffb7304400 r \_\_ksymtab\_avenrun

ffffffffb73237b8 r \_\_kcrctab\_avenrun

ffffffffb733523a r \_\_kstrtab\_avenrun

ffffffffb7803750 B avenrun

 4)**使用 kallsyms\_lookup\_name() 函数**：  
 是在kernel/kallsyms.c文件中定义的，要使用它必须启用CONFIG\_KALLSYMS编译内核。  
 kallsyms\_lookup\_name()接受一个字符串格式内核函数名，返回那个内核函数的地址。  
        kallsyms\_lookup\_name("avenrun ");

avenrun = ffffffffb7803750

总结：3 4方法可用与实际测试一致，其它仅供参考原因未明

## 7.10 x86添加系统调用

系统调用的两种方式：

1 通过libc

2

系统调用的一般流程描述



以fork()为例说明

User\_space

Main()

{

Fork();

}

Libc.so

Fork()

{

Movl 2 ,%eax;// 2 把fork对应的系统调用号放进到eax里面

Int $0x80 //3 通过0x80陷入到内核

}

实例分析系统调用规则（getuid为例）：

1 静态编译使用gdb调试 gcc -g -static fork.c



2 启动gdb 开始调试



3 开始调用getuid()

Mov $0x66,%eax 将系统调用号送进寄存器%eax

然后syscall 指定地址的软中断进入kernel



【4】

X86添加系统调用

1 linux-4.19.122/arch/x86/entry/syscalls/syscall\_64.tbl



335 common myaudit \_\_x64\_sys\_myaudit

【2】 add系统调用文件 /linux-4.19.122/arch/x86/kernel/



【3】修改Makefile /linux-4.19.122/arch/x86/kernel/Makefile



【4】添加函数声明 /linux-4.19.122/include/linux/syscalls.h



【5】拦截相关的系统调用 /linux-4.19.122/arch/x86/entry/common.c

如图添加了2 3 39 56 57 59 系统调用拦截

（主要意思是：当发生2号系统调用，调用自己的内核模块）



## 7.11 ARM添加系统调用

1 系统调用号定义 arch/arm/include/uapi/asm/unistd.h





2 具体系统调用实现的接口 /arch/arm/kernel/Call.S



3 添加函数的具体实现方式 arch/arm/kernel/sys\_arm.c



/\*

SYSCALL\_DEFINE2(getpuid, pid\_t \_\_user \*, pid, uid\_t \_\_user \*,uid)

{

if(pid == NULL && uid == NULL)

return -EINVAL;

if(pid!=NULL)

\*pid = task\_tpid\_vnr(current);

if(uid != NULL)

\*uid = from\_kuid\_munged(current\_user\_ns(),corrent\_uid());

return 0;

}

\*/

## 7.12 调试中断使用的方法和技巧

1 查看系统有的中断（仅仅记录当前注册的中断）详细的查看/proc/stat



2 安装驱动代码之后

LinuxTestCode/module/14\_interrupt\_/interrupt\_upper



问题：如何手动触发一号中断？

3 中断下半部分处理不紧急的事情

LinuxTestCode/module/14\_interrupt\_/interrupt\_lower

添加中断下半部分处理

## 7.13 do\_page\_fault()调试技巧

【1】如何查看do\_page\_fault发生次数 ：ps -o majflt,minflt -p 1

发成缺页中断后，执行了那些操作？  
当一个进程发生缺页中断的时候，进程会陷入内核态，执行以下操作：  
1、检查要访问的虚拟地址是否合法  
2、查找/分配一个物理页  
3、填充物理页内容（读取磁盘，或者直接置0，或者啥也不干）  
4、建立映射关系（虚拟地址到物理地址）  
重新执行发生缺页中断的那条指令  
如果第3步，需要读取磁盘，那么这次缺页中断就是majflt，否则就是minflt。



【2】或者修改源码去计数

每次调用do\_page\_fault全局变量加一 /linux-4.19.122/arch/x86/mm/ fault.c





添加一个声明linux-4.19.122/include/linux/mm.h



导出全局变量内核都可以使用linux-4.19.122/kernel/kallsyms.c



## 7.14 缺页异常的一些总结

【1】什么时机引起缺页异常？

虚拟地址到物理地址转换的过程中，

1 CPU访问逻辑地址（0，1024）->缺页中断机构查找是否在内存

1. 用户态的程序在执行，自身占用4G的虚拟内存空间，并非4G的大小都在内存中，需要的话降内存调入内存使用。程序访问的页面不存在，产生缺页中断，自身放进阻塞队列。页面请求成功待，返回代码继续执行。
2. 页面置换算法

如果存在空闲页表（直接放进内存使用）

如果不存在空闲也变（置换不常用的内存到外存）

arch/arm/mm/fault.c

static int [**\_\_kprobes**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/__kprobes)

[**do\_page\_fault**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/do_page_fault)(unsigned [**long**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/long) addr, unsigned int fsr, struct [**pt\_regs**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/pt_regs) \*regs)

{

struct [**task\_struct**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/task_struct) \*[**tsk**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/tsk);

struct [**mm\_struct**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/mm_struct) \*[**mm**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/mm);

int [**fault**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/fault), [**sig**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/sig), [**code**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/code);

unsigned int flags = [**FAULT\_FLAG\_ALLOW\_RETRY**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/FAULT_FLAG_ALLOW_RETRY) | [**FAULT\_FLAG\_KILLABLE**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/FAULT_FLAG_KILLABLE);

if ([**notify\_page\_fault**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/notify_page_fault)(regs, fsr))

return 0;

[**tsk**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/tsk) = [**current**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/current);

[**mm**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/mm) = [**tsk**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/tsk)->[**mm**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/mm);//需要判断是内核或者用户的page\_fault

*/\**

*\* If we're in an interrupt or have no user*

*\* context, we must not take the fault..*

*\*/*

if ([**faulthandler\_disabled**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/faulthandler_disabled)() || ![**mm**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/mm))//判断内核发生了 缺页异常

goto [**no\_context**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/no_context);//强制执行检测

｝

用户态：

[**\_\_do\_page\_fault**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/__do_page_fault)([**mm**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/mm), addr, fsr, flags, [**tsk**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/tsk));

CR2是页故障线性地址寄存器，保存最后一次出现页故障的全32位线性地址

address = read\_cr2();

内核态：

[**\_\_do\_kernel\_fault**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/__do_kernel_fault)([**mm**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/mm), addr, fsr, regs);

## 7.15 逻辑地址 虚拟地址 线性地址 物理地址

线性地址(或者叫做虚拟地址) = 逻辑地址+段基址

Linux中逻辑地址等于线性地址。为什么这么说呢？因为Linux所有的段（用户代码段、用户数据段、内核代码段、内核数据段）的线性地址都是从 0x00000000 开始，长度4G，这样 线性地址=逻辑地址+ 0x00000000，也就是说逻辑地址等于线性地址了。

## 7.16 find\_vma()理解

代码地址：LinuxTestCode/module/16\_find\_vma/



一个进程的地址空间管理,有多个vma\_area\_struct结构的虚拟区间

Task\_stauct->mm\_struct->vma\_area\_struct()

vma\_area\_struct()

vma\_area\_struct()

vma\_area\_struct()



使用crash调试

【1】task <pid>



【2】mm\_struct \*mm首地址：mm = 0xffff93aa36280f80,



【3】通过查找struct mm\_struct 0xffff93aa36280f80获取mmap的首地址



【4】查找第一个vm地址：struct vm\_area\_struct 0xffff93aa001123c0



【5】





【6】个人理解VMA是段机制，连续的具有相同属性，是物理页面的整数倍

用户空间的maps参数



内核空间地址求出的vma地址



## 7.17 malloc vmalloc kmalloc kmalloc get\_free\_page()区别联

32位系统：用户3G <->内核1G

用户：3G的虚拟内存空间，如果物理内存4G怎么办？

内核空间：假如0xc000 0000- 0xffffffff一一映射物理地址，只能访问前1G的物理空间，后面的地址将无法访问。



需要思考如何映射才能满足映射？（大于1G物理内存如何映射才好）

在x86结构中，三种类型的区域如下：

**ZONE\_DMA**      内存开始的16MB

**ZONE\_NORMAL**     16MB~896MB

**ZONE\_HIGHMEM**       896MB ~ 结束（128M）

内核将高端内存划分为3部分：

VMALLOC\_START~VMALLOC\_END、

KMAP\_BASE~FIXADDR\_START

FIXADDR\_START~4G



如何使用128M的空间映射 剩余所有的物理内存呢？

128M空间如何使用呢！！可以临时建立映射关系，使用完毕之后归还。

比如：要访问2G的1M的空间地址是0×80000000 ~ 0x800FFFFF

首先需要建立一个映射表：从高端地址寻求映射关系

0xF8700000 0x80000000



使用之后释放空间，供下一个使用者使用。（可能会出现使用一致占用）

从上面的描述，我们可以知道**高端内存的最基本思想**：借一段地址空间，建立临时地址映射，用完后释放，达到这段地址空间可以循环使用，访问所有物理内存

64位系统：用户512G <->内核2^48-512G（2^48-2^39=2^）

内核空间很大不需要映射就可以取边所有的物理空间，不存在高端映射

0x0000000000-0x8000000000 (属于虚拟地址用户空间)

用户空间：

Malloc

malloc过程中会创建一个heap,首次创建是0x0258f000-0x0256e000=0x21000(33\*page大小)

等这个使用完之后，系统再次分配一个00x21000大小个虚拟内存

如下图所示：

0256e000-0258f000 rw-p 00000000 00:00 0 [heap]

0256e000-025b0000 rw-p 00000000 00:00 0 [heap]

0256e000-025d1000 rw-p 00000000 00:00 0 [heap]

内核空间

Vmalloc

Kmalloc

Get\_free\_page （只能得到连续的页面）

**kmalloc()和vmalloc()介绍  
kmalloc()  
用于申请较小的、连续的物理内存  
1. 以字节为单位进行分配，在<linux/slab.h>中  
2. void \*kmalloc(size\_t size, int flags) 分配的内存物理地址上连续，虚拟地址上自然连续  
3. gfp\_mask标志**：什么时候使用哪种标志？如下：  
———————————————————————————————-  
情形 相应标志  
———————————————————————————————-  
进程上下文，可以睡眠 GFP\_KERNEL  
进程上下文，不可以睡眠 GFP\_ATOMIC  
中断处理程序 GFP\_ATOMIC  
软中断 GFP\_ATOMIC  
Tasklet GFP\_ATOMIC  
用于DMA的内存，可以睡眠 GFP\_DMA | GFP\_KERNEL  
用于DMA的内存，不可以睡眠 GFP\_DMA | GFP\_ATOMIC  
———————————————————————————————-  
**4. void kfree(const void \*ptr)  
释放由kmalloc()分配出来的内存块**

**vmalloc()  
用于申请较大的内存空间，虚拟内存是连续的**1. **以字节为单位进行分配**，在<linux/vmalloc.h>中  
2. **void \*vmalloc(unsigned long size) 分配的内存虚拟地址上连续，物理地址不连续**3. 一般情况下，只有硬件设备才需要物理地址连续的内存，因为硬件设备往往存在于MMU之外，根本不了解虚拟地址；但为了性能上的考虑，内核中一般使用 kmalloc()，而只有在需要获得大块内存时才使用vmalloc()，例如当模块被动态加载到内核当中时，就把模块装载到由vmalloc()分配 的内存上。  
4.**void vfree(void \*addr)，这个函数可以睡眠，因此不能从中断上下文调用。**

**malloc(), vmalloc()和kmalloc()区别**[\*]kmalloc和vmalloc是分配的是内核的内存,malloc分配的是用户的内存  
[\*]kmalloc保证分配的内存在物理上是连续的,vmalloc保证的是在虚拟地址空间上的连续,malloc不保证任何东西(这点是自己猜测的,不一定正确)  
[\*]kmalloc能分配的大小有限,vmalloc和malloc能分配的大小相对较大  
[\*]内存只有在要被DMA访问的时候才需要物理上连续  
[\*]vmalloc比kmalloc要慢

***kmalloc()与vmalloc()的区别***

***kmalloc()与vmalloc() 都是在内核代码中提供给其他子系统用来分配内存的函数，但二者有何区别？***

***从前面的介绍已经看出，这两个函数所分配的内存都处于内核空间，即从3GB～4GB；但位置不同，kmalloc()分配的内存处于3GB～high\_memory之间，而vmalloc()分配的内存在VMALLOC\_START～4GB之间，也就是非连续内存区。一般情况下在驱动程序中都是调用kmalloc()来给数据结构分配内存，而vmalloc()用在为活动的交换区分配数据结构，为某些I/O驱动程序分配缓冲区，或为模块分配空间，例如在include/asm-i386/module.h中定义了如下语句：***

***#define module\_map(x)           vmalloc(x)***

***其含义就是把模块映射到非连续的内存区。***

***与kmalloc()和vmalloc()相对应，两个释放内存的函数为kfree()和vfree()。***

## 7.18 page在内核如何被管理

查看系统页面的大小

K89441AA1:/ # getconf PAGESIZE



WARNING:"mem\_map" xxxxxxxxxxxxxxx undefined!

Mem\_map定义在 include/linux/mmzone.h

#ifndef CONFIG\_DISCONTIGMEM

/\* The array of struct pages - for discontigmem use pgdat->lmem\_map \*/

extern struct page \*mem\_map;

#endif

暂时不能使用mem\_map 打印page信息。

## 7.19 两个进程的虚拟地址一样，为什么访问的物理地址不一样？从操作系统角度来说

以如下的：27602和36298进程为例

从如下两个的进程可以看到：

Pgd(页全局目录不一样)，进行虚拟->物理转换过程中，需要把pdg给cr3寄存器。



Pid 27602 进程如下特点：







Pid 36298进程如下特点：







## 7.21 搞明白一个进程所占用的物理内存是哪个页面

使用代码读取参数指定进程的参数

LinuxTestCode/module/20\_atomic\_read









Hiwater\_rss = 154个页面（154\*4=616k）









total\_rss << (PAGE\_SHIFT-10),

task\_mmu.c

hiwater\_rss = total\_rss = anon + file + shmem;

void task\_mem(struct seq\_file \*m, struct mm\_struct \*mm)

{

unsigned long anon, file, shmem;

anon = get\_mm\_counter(mm, MM\_ANONPAGES);

file = get\_mm\_counter(mm, MM\_FILEPAGES);

shmem = get\_mm\_counter(mm, MM\_SHMEMPAGES);

}

static inline unsigned long get\_mm\_counter(struct mm\_struct \*mm, int member)

{

long val = atomic\_long\_read(&mm->rss\_stat.count[member]);

#ifdef SPLIT\_RSS\_COUNTING

/\*

\* counter is updated in asynchronous manner and may go to minus.

\* But it's never be expected number for users.

\*/

if (val < 0)

val = 0;

#endif

return (unsigned long)val;

Htop 可以查看

/proc/<pid>/status

top -p 2913

<https://blog.csdn.net/keplerpig/article/details/72867016>

https://www.cnblogs.com/freeweb/p/5407105.html

具体是哪个代码 表达的是 物理内存的参数

明天主要写下 top源码中 计算物理内存的方法？

搞明白一个结构体inode\_operations是什么作用？

很好的文章需要文件系统知识才能解答!

<http://www.voidcn.com/article/p-cwfpoeou-bpg.html>

## 7.22 /proc/文件系统相关的操作结构体

Include/linux/seq\_file.h

struct seq\_operations {

void \* (\*start) (struct seq\_file \*m, loff\_t \*pos);

void (\*stop) (struct seq\_file \*m, void \*v);

void \* (\*next) (struct seq\_file \*m, void \*v, loff\_t \*pos);

int (\*show) (struct seq\_file \*m, void \*v);

};

Include/linux/seq\_file.h

struct seq\_file {

char \*buf;

size\_t size;

size\_t from;

size\_t count;

size\_t pad\_until;

loff\_t index;

loff\_t read\_pos;

u64 version;

struct mutex lock;

const struct seq\_operations \*op;

int poll\_event;

const struct file \*file;

void \*private;

};

## 7.23 总结下原子操作atomic\_XXX

## 7.24 pfn\_to\_page

LinuxTestCode/module/21\_pfn\_to\_page

## 7.25 ps –o xxx,xxx 源码理解

Task\_srtuct 中包含缺页中断的次数

LinuxTestCode/module/22\_no\_page\_statistic

## 7.26 线程和内核线程

线程没有内存空间

内核线程也是没有内核空间的

问题：子线程是否和主线程共用一个task\_struct?

## 7.27 CPU上的运行队列思考

GraphicalLinuxSubsystem.xlsx【rq结构体关系图】

每个CPU都有自己的一个rq队列，并且只能运行一个进程，靠切换获得CPU的使用权。

## 7.28 手动读取内核的 loadavg

首先明确一点 avenrun是内核的一个全局变量

获取全局变量的方法：

1. 代码见/LinuxTestCode/module/12\_load\_monitor

ptr\_avenrun = (void \*)kallsyms\_lookup\_name("avenrun");//获取内核变量的地址

kernel/sched/loadavg.c

void get\_avenrun(unsigned long \*loads, unsigned long offset, int shift)

{

loads[0] = (avenrun[0] + offset) << shift;

loads[1] = (avenrun[1] + offset) << shift;

loads[2] = (avenrun[2] + offset) << shift;

}

avenrun 首地址保存了三个数组分别是最近一分钟、最近5分钟、最近15分钟的负载





410/2^11 ->0.21

366/2^11 ->0.18

163/2^11 ->0.08

## 7.29 CPU负载如何计算的

【1】计算有多少个CPU



【2】计算CPU总的使用情况



总的cpu时间totalCpuTime = user + nice + system + idle + iowait + irq + softirq + stealstolen  +  guest



【3】计算进程总的使用时间



1. 单核CPU使用率计算

## 7.30 尝试分析/proc/buddyinfo

## 7.31 尝试分析/proc/slabinfo

<https://blog.csdn.net/lukuen/article/details/6935068>

static int slab\_show(struct seq\_file \*m, void \*p)

{

struct kmem\_cache \*s = list\_entry(p, struct kmem\_cache, list);//求出首地址

cache\_show(s, m);

}

static void cache\_show(struct kmem\_cache \*s, struct seq\_file \*m)

{

struct slabinfo sinfo;

memset(&sinfo, 0, sizeof(sinfo));

get\_slabinfo(s, &sinfo);

memcg\_accumulate\_slabinfo(s, &sinfo);

seq\_printf(m, "%-17s %6lu %6lu %6u %4u %4d",

cache\_name(s), sinfo.active\_objs, sinfo.num\_objs, s->size,

sinfo.objects\_per\_slab, (1 << sinfo.cache\_order));

seq\_printf(m, " : tunables %4u %4u %4u",

sinfo.limit, sinfo.batchcount, sinfo.shared);

seq\_printf(m, " : slabdata %6lu %6lu %6lu",

sinfo.active\_slabs, sinfo.num\_slabs, sinfo.shared\_avail);

slabinfo\_show\_stats(m, s);

seq\_putc(m, '\n');

}

## 7.32 linux函数链表总结

源码：LinuxTestCode/module/23\_list\_op





## 7.33 slab的总结问题

见：GraphicalLinuxSubsystem.xlsx【slab图解】



slabinfo显示的是16和申请的4不一致为什么？

slab之针对内核空间与用户空间无关

## 7.34 自定义内核全局变量

LinuxTestCode/module/28\_export\_my\_global\_var

## 7.35 覆盖打印输出

LinuxTestCode/user/10\_over\_print

## 7.36 打印所有的kernel所有的slab缓存

LinuxTestCode/module/27\_print\_all\_slab\_info

slab最初由XXX结构体建立一个管理结构，以后使用函数kmem\_cache\_create()，向结构体链表中插入slab.

## 7.36 per\_cpu实验

## 7.36 查看进程使用那个cpu工作

【1】taskset -c -p 5



【2】htop 添加 columns-processor显示进程在那个cpu上





【3】top

也可以但是没有实验成功

1. ps -eo pid,args,psr
2. ps 命令的神奇之处在于可以自定义输出格式

查看当前系统进程的uid,pid,stat,pri, 以uid号排序.  
ps -eo pid,stat,pri,uid –sort uid  
  
查看当前系统进程的user,pid,stat,rss,args, 以rss排序.  
ps -eo user,pid,stat,rss,args –sort rss

Ps -o(自定义)-e(所有进程) （自定义显示的内容）

## 7.36 打印进程所在的CPU

LinuxTestCode/module/30\_print\_cpu\_process

## 7.36 关于调度器的/proc/xx/sched信息解读

参考地址：https://blog.csdn.net/wudongxu/article/details/8574755

se.exec\_start       :    2781299327.397282  //此进程最近被调度到的开始执行时刻（这个值是每次update\_curr都进行更新）

se.vruntime        :       3144603.079903  //虚拟运行时间

se.sum\_exec\_runtime:       2843625.998498  //累计运行的物理时间时间

se.wait\_start       :             0.000000  //最近一次当前进程被入队的时刻

se.sleep\_start      :             0.000000  //此进程最近一次被从队列里取出，并被置S状态的时刻

se.block\_start      :             0.000000  //此进程最近一次被从队列里取出，并被置D状态的时刻

se.sleep\_max      :             0.000000  //最长处于S状态时间

se.block\_max      :             0.000000  //最长处于D状态时间

se.exec\_max       :             1.004266  //最长单次执行时间

se.slice\_max       :           998.456300  //曾经获得时间片的最长时间

se.wait\_max       :             0.455235  //最长在就绪队列里的等待时间

se.wait\_sum       :            15.615407  //累计在就绪队列里的等待时间

se.wait\_count      :                 3147  //累计等待次数

se.iowait\_sum      :           215.825267  //io等待时间

se.iowait\_count     :                   67 //io等待次数  io\_schedule调用次数

sched\_info.bkl\_count:                    0  //此进程大内核锁调用次数

se.nr\_migrations    :                    0 //需要迁移当前进程到其他cpu时累加此字段

se.nr\_migrations\_cold:                    0

se.nr\_failed\_migrations\_affine:           194  //进程设置了cpu亲和，进程迁移时检查失败的次数

se.nr\_failed\_migrations\_running:           0

se.nr\_failed\_migrations\_hot:               0  //当前进程因为是cache hot导致迁移失败的次数

se.nr\_forced\_migrations :                  0  //在当前进程cache hot下，由于负载均衡尝试多次失败，强行进行迁移的次数

se.nr\_wakeups         :                 0  //被唤醒的累计次数（从不可运行到可运行）

se.nr\_wakeups\_sync     :                0  //同步唤醒次数，即a唤醒b，a立刻睡眠，b被唤醒的次数

se.nr\_wakeups\_migrate  :                 0 //被唤醒得到调度的当前cpu，不是之前睡眠的cpu的次数

se.nr\_wakeups\_local     :                0 //被本地唤醒的次数（唤醒后在当前cpu上执行）

se.nr\_wakeups\_remote   :                0 //非本地唤醒累计次数

se.nr\_wakeups\_affine    :                0 //考虑了任务的cache亲和性的唤醒次数

se.nr\_wakeups\_affine\_attempts:            0

se.nr\_wakeups\_passive  :                    0

se.nr\_wakeups\_idle     :                    0

avg\_atom            :           903.886204 //本进程平均耗时sum\_exec\_runtime/ nr\_switches

avg\_per\_cpu           :             0.000001

nr\_switches            :                 3146 //主动切换和被动切换的累计次数

nr\_voluntary\_switches   :                    0 //主动切换次数（由于prev->state为不可运行状态引起的切换）

nr\_involuntary\_switches  :                 3146 //被动切换次数

se.load.weight          :                 1024  //该se的load

policy                 :                    0  //调度策略 normal

prio                   :                  120  //优先级(nice=0)

clock-delta             :                   51

## 7.36 关于调度器的/proc/xx/stat信息解读

源码打印：/fs/proc/arrary.c

do\_task\_stat()

28733 (cpu\_test) R 5573 28733 5573 34824 5573 4202496 176 0 0 0 1797172 80 0 20 0 1 0 2782750364005888 82 184467440737095516154194304 4195884 140733625322688 140733625322136 41954610 0 0 0 0 0 0 17 3 00 0 0 0

seq\_printf(m, "%d (%s) %c %d %d%d %d %d %u %lu \

%lu %lu %lu %lu %lu %ld %ld %ld %ld %d 0 %llu %lu %ld %lu %lu %lu %lu %lu \

%lu %lu %lu %lu %lu %lu%lu %lu %d %d %u %u %llu %lu %ld\n",

1 .pid\_nr\_ns(pid, ns), //pid

2 .tcomm, //进程名

3 .state, //进程状态

4 .ppid, //父进程号

5 .pgid, //线程组ID

6 .sid, //会话组ID

7 .tty\_nr, //该进程的tty终端的设备号，INT（34817/256）=主设备号，（34817-主设备号）=次设备号

8 .tty\_pgrp, //终端的进程组号，当前运行在该进程所在终端的前台进程(包括shell应用程序)的PID

9 .task->flags, //进程标志位，查看该进程的特性(定义在/include/kernel/sched.h中)

10.min\_flt, //累计进程的次缺页数（Copy on Write页和匿名页）

11.cmin\_flt, //该进程所有的子进程发生的次缺页的次数

12.maj\_flt, //主缺页数（从映射文件或交换设备读入的页面数）

13.cmaj\_flt, //该进程所有的子进程发生的主缺页的次数

14.cputime\_to\_clock\_t(utime), //该进程在用户态运行的时间，单位为jiffies

15.cputime\_to\_clock\_t(stime), //该进程在核心态运行的时间，单位为jiffies

16.cputime\_to\_clock\_t(cutime), //该进程所有的子进程在用户态运行的时间总和，单位为jiffies

17.cputime\_to\_clock\_t(cstime), //该进程所有的子进程在内核态运行的时间的总和，单位为jiffies

18.priority, //进程的优先级

19.nice, //进程的静态优先级

20.num\_threads, //该进程所在的线程组里线程的个数

21.//0

22.start\_time, //该进程创建的时间

23.vsize, //该进程的虚拟地址空间大小

24.mm ? get\_mm\_rss(mm) : 0, //该进程当前驻留物理地址空间的大小

25.rsslim, //该进程能驻留物理地址空间的最大值

26.mm ? (permitted ? mm->start\_code : 1) : 0, //该进程在虚拟地址空间的代码段的起始地址

27.mm ? (permitted ? mm->end\_code : 1) : 0, //该进程在虚拟地址空间的代码段的结束地址

28.(permitted && mm) ? mm->start\_stack : 0, //该进程在虚拟地址空间的栈的结束地址

29.esp, //esp(32 位堆栈指针) 的当前值,与在进程的内核堆栈页得到的一致

30.eip, //指向将要执行的指令的指针, EIP(32位指令指针)的当前值

31.task->pending.signal.sig[0] & 0x7fffffffUL, //待处理信号的位图，记录发送给进程的普通信号

32.task->blocked.sig[0] & 0x7fffffffUL, //阻塞信号的位图

33.sigign      .sig[0] & 0x7fffffffUL,  //忽略的信号的位图

34.sigcatch    .sig[0] & 0x7fffffffUL,  //被捕捉的信号的位图

35.wchan, //如果该进程是睡眠状态，该值给出调度的调用点

36.0UL,

37.0UL,

38.task->exit\_signal, //该进程结束时，向父进程所发送的信号

39.task\_cpu(task), //运行在哪个CPU上

40.task->rt\_priority, //实时进程的相对优先级别

41.task->policy,  //进程的调度策略

42.(unsigned long long)delayacct\_blkio\_ticks(task),

43.cputime\_to\_clock\_t(gtime),

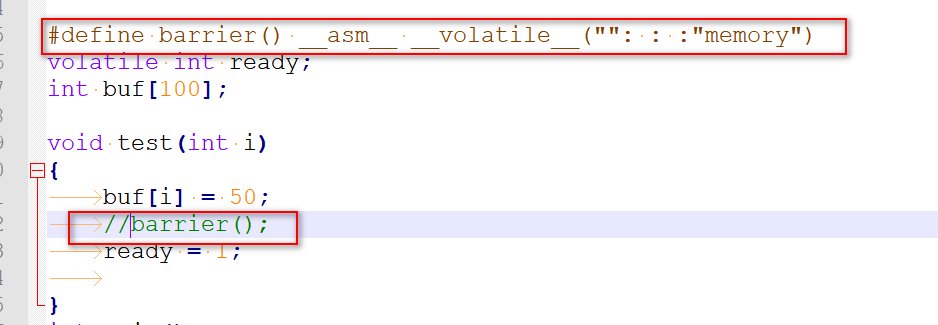
44.cputime\_to\_clock\_t(cgtime));

## 7.36 关于内存屏障的一些理解和volatile关系

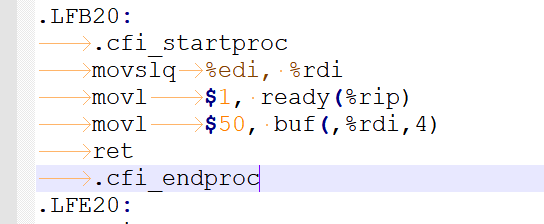
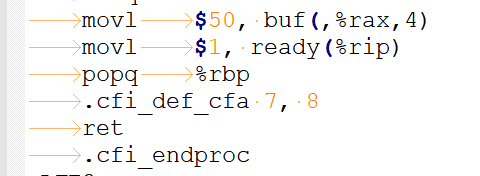
LinuxTestCode/user/11\_memory barrier\_optimization

编译器编译代码和CPU执行指令都会 都会对代码重排优化处理，避免这样的事情发生，需要加入屏障，

作用：保障屏障之前的代码 先执行（不管是否加载了优化）



下图所示开启优化和关闭优化GCC对代码的优化，指令先后顺序变化。



关于arm64内存屏障定义：

由宏定义转为汇编语言

#define mb() dsb(sy)

#define rmb() dsb(ld)

#define wmb() dsb(st)

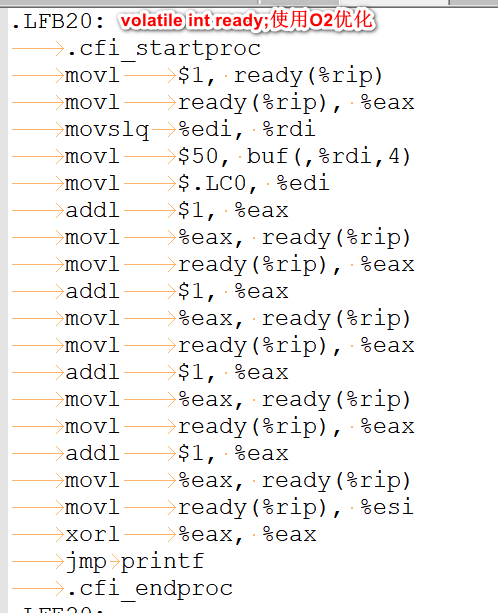
#define isb(option) \_\_asm\_\_ \_\_volatile\_\_ ("isb " #option : : : "memory")

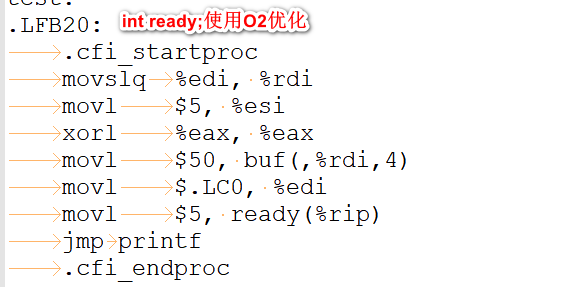
#define dsb(option) \_\_asm\_\_ \_\_volatile\_\_ ("dsb " #option : : : "memory")

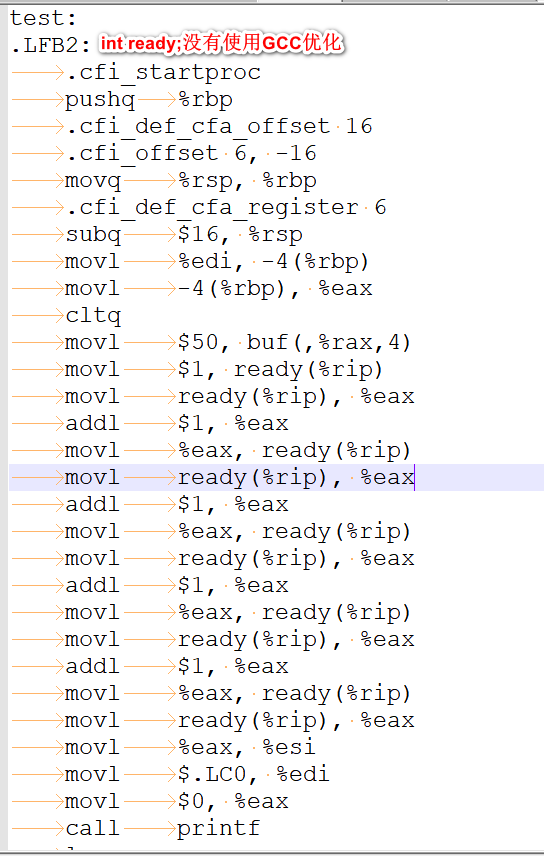
#define dmb(option) \_\_asm\_\_ \_\_volatile\_\_ ("dmb " #option : : : "memory")

**Volatile：**

1. **强制CPU从内存都数据，不要从寄存器读数据**
2. **告诉编译器和CPU不要优化这个数据。**







## 7.36 C语言内嵌汇编

/LinuxTestCode/user/12\_c\_inline\_asm

/\*

例子：#define barrier() \_\_asm\_\_ \_\_volatile\_\_("": : :"memory")

#define \_\_asm\_\_ asm

#define \_\_volatile\_\_ volatile

\_\_asm\_\_(汇编语句模板: 输出部分: 输入部分: 破坏描述部分)

\*/

## 7.36 添加个实验自定义一个函数让1和2号进程呼叫进程。

https://blog.csdn.net/m0\_37598340/article/details/86606334?utm\_medium=distribute.pc\_relevant\_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase&depth\_1-utm\_source=distribute.pc\_relevant\_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase

init初始化流程

init/mian.c

asmlinkage \_\_visible void \_\_init start\_kernel(void)

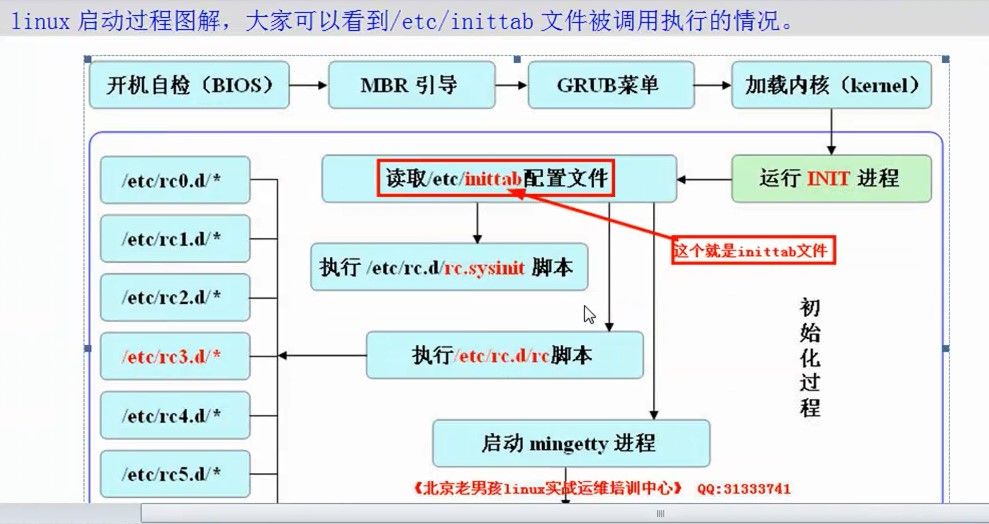
/\* Do the rest non-\_\_init'ed, we're now alive \*/

rest\_init();

kernel\_thread(kernel\_init(回调), NULL, CLONE\_FS);

ret = run\_init\_process(execute\_command);

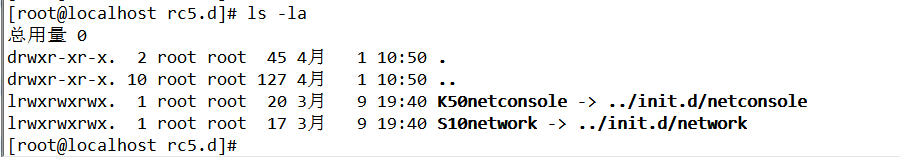
do\_execve(getname\_kernel(init\_filename),



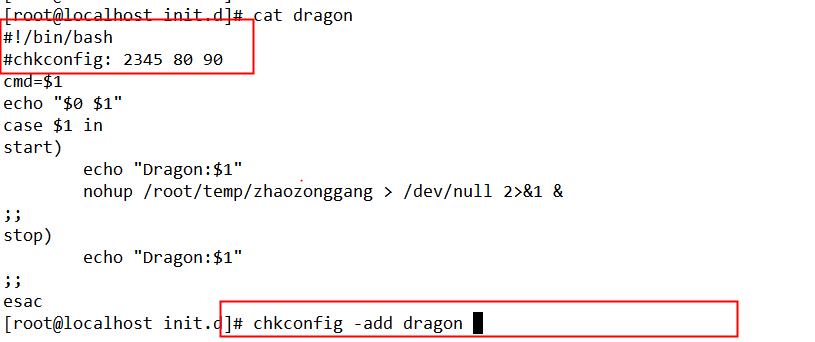
考虑过系统的启动进程，最好不要修改：

自己添加一个系统服务





代码如下添加：使用命令添加自启动



示例，auto\_run的前三行如下：

#!/bin/sh

#chkconfig: 2345 80 90

#description:auto\_run

第一行，告诉系统使用的shell,所以的shell脚本都是这样。

第二行，chkconfig后面有三个参数2345,80和90告诉chkconfig程序，需要在rc2.d~rc5.d目录下，创建名字为S80auto\_run的文件连接，连接到/etc/rc.d/init.d目录下的的auto\_run脚本。第一个字符是S，系统在启动的时候，运行脚本auto\_run，就会添加一个start参数，告诉脚本，现在是启动模式。同时在rc0.d和rc6.d目录下，创建名字为K90auto\_run的文件连接，第一个字符为K，个系统在关闭系统的时候，会运行auto\_run，添加一个stop，告诉脚本，现在是关闭模式。

注意上面的三行是中，地二，第三行是必须的，否则在运行chkconfig --add auto\_run时，会报错。

1、这些链接文件前面为什么会带一个Kxx或者Sxx呢？

是这样的，带K的表示停止(Kill)一个服务，S表示开启(Start)的意思

2、K和S后面带的数字呢？干什么用的？

这个我开始的时候还以为是排列起来好看或者数数用呢(是不是很幼稚？)。后来发现不是的。它的作用是用来排序，就是决定这些脚本执行的顺序，数值小的先执行，数值大的后执行。很多时候这些执行顺序是很重要的，比如要启动Apache服务，就必须先配置网络接口，不然一个没有IP的机子来启动http服务那岂不是很搞笑。。。

2、这时我才想起来原来曾经无数用过的/etc/rc.d/init.d/network  restart命令。

传S时相当于执行了/etc/rc.d/init.d/xxx  start这条命令，

K就相当于/etc/rc.d/init.d/xxx  stop了。

## 7.36 内核创建线程的方法

LinuxTestCode/module/31\_kernel\_thread\_\_kthread\_creat

【1】kernel\_thread(4.0以后的内核不可以直接使用，没有导出)

【2】kthread\_run/kthread\_create（最后调用kthread\_create\_on\_node）

#define [**kthread\_create**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/kthread_create)([**threadfn**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/threadfn), data, namefmt, arg...) \

[**kthread\_create\_on\_node**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/kthread_create_on_node)([**threadfn**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/threadfn), data, [**NUMA\_NO\_NODE**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/NUMA_NO_NODE), namefmt, ##arg)

#define [**kthread\_run**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/kthread_run)([**threadfn**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/threadfn), data, namefmt, ...) \

({

struct [**task\_struct**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/task_struct) \*\_\_k \

= [**kthread\_create**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/kthread_create)([**threadfn**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/threadfn), data, namefmt, ## \_\_VA\_ARGS\_\_); \

if (![**IS\_ERR**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/IS_ERR)(\_\_k)) \

[**wake\_up\_process**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/wake_up_process)(\_\_k);

\_\_k; \

})

【30】kthread\_create\_on\_node

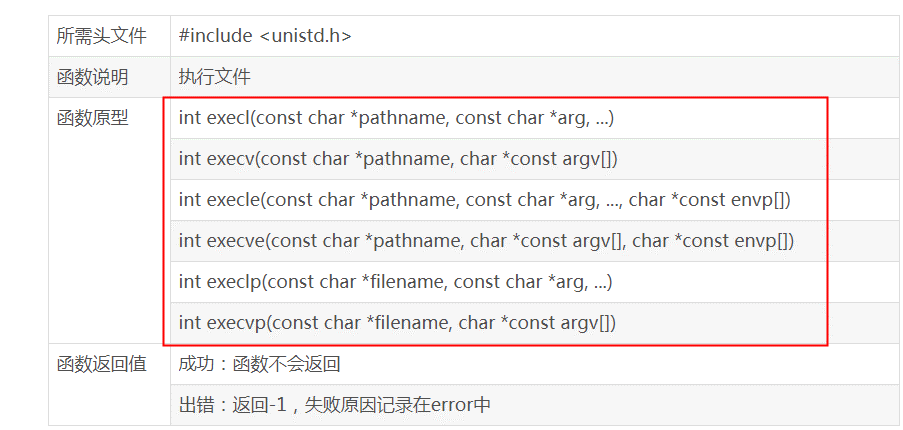
## 7.36 exec函数详解

LinuxTestCode/user/13\_exec\_group

<https://blog.csdn.net/gatieme/article/details/51594439>

<https://blog.csdn.net/yychuyu/article/details/80173039>

所需头文件



解释说明：

这6 个函数在函数名和使用语法的规则上都有细微的区别，下面就可执行文件查找方式、参数表传递方式及环境变量这几个方面进行比较说明。

①    查找方式：上表其中前4个函数的查找方式都是完整的文件目录路径（pathname），而最后2个函数（也就是以p结尾的两个函数）可以只给出文件名，系统就会自动从环境变量“$PATH”所指出的路径中进行查找。

前4个取路径名做参数，后两个则取文件名做参数。

当指定filename做参数时：

a. 如果filename中包含/，则将其视为路径名

b. 否则就按PATH环境变量搜索可执行文件。

②    参数传递方式：exec函数族的参数传递有两种方式，一种是逐个列举(l)的方式，而另一种则是将所有参数整体构造成指针数组(v)进行传递。

在这里参数传递方式是以函数名的第5位字母来区分的，字母为“l”（list）的表示逐个列举的方式，字母为“v”（vertor）的表示将所有参数整体构造成指针数组传递，然后将该数组的首地址当做参数传给它，数组中的最后一个指针要求是NULL。读者可以观察execl、execle、execlp的语法与execv、execve、execvp的区别。

③    环境变量：exec函数族使用了系统默认的环境变量，也可以传入指定的环境变量。这里以“e”（environment）结尾的两个函数execle、execve就可以在envp[]中指定当前进程所使用的环境变量替换掉该进程继承的所以环境变量。

（3）PATH环境变量说明

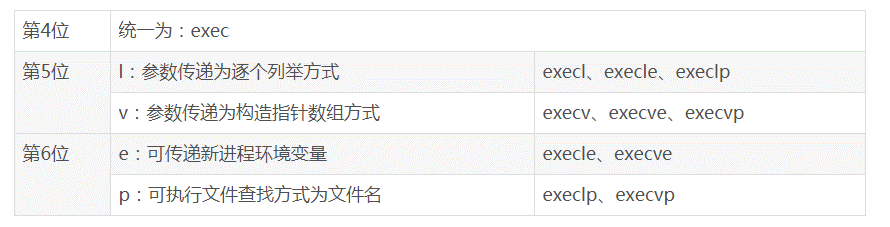
PATH环境变量包含了一张目录表，系统通过PATH环境变量定义的路径搜索执行码，PATH环境变量定义时目录之间需用用“:”分隔，以“.”号表示结束。PATH环境变量定义在用户的.profile或.bash\_profile中，下面是PATH环境变量定义的样例，此PATH变量指定在“/bin”、“/usr/bin”和当前目录三个目录进行搜索执行码。

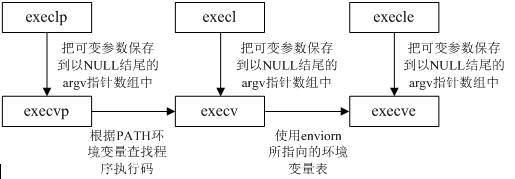
PATH=/bin:/usr/bin:.

export $PATH

（4）进程中的环境变量说明

    在Linux中，Shell进程是所有执行码的父进程。当一个执行码执行时，Shell进程会fork子进程然后调用exec函数去执行执行码。Shell进程堆栈中存放着该用户下的所有环境变量，使用execl、execv、execlp、execvp函数使执行码重生时，Shell进程会将所有环境变量复制给生成的新进程；而使用execle、execve时新进程不继承任何Shell进程的环境变量，而由envp[]数组自行设置环境变量。

（5）



## 7.36 CPU上下文切换

进程上下文切换

需要保存的环境变量有哪些？

1 寄存器和程序计数器保存在内存中

2 寻找下一个进程，恢复所有下文信息->寄存器和程序计数器

3 返回程序计数器地址，继续执行程序

线程上下文切换

context\_switch(struct rq \*rq, struct task\_struct \*prev,

struct task\_struct \*next, struct pin\_cookie cookie)

{

struct mm\_struct \*mm, \*oldmm;

prepare\_task\_switch(rq, prev, next);

mm = next->mm;//待切换进程的mm赋值给现在的mm

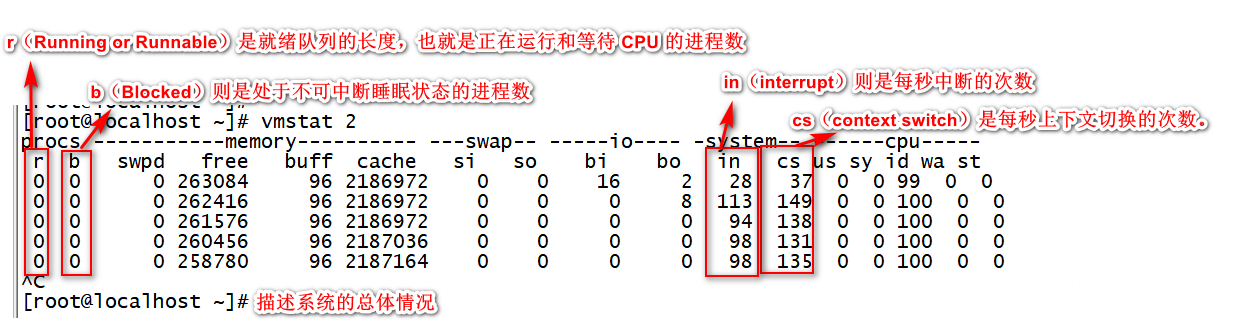
oldmm = prev->active\_mm;//之前的active\_mm如何处理

}

不同进程间的线程切换

相同进程间的线程切换

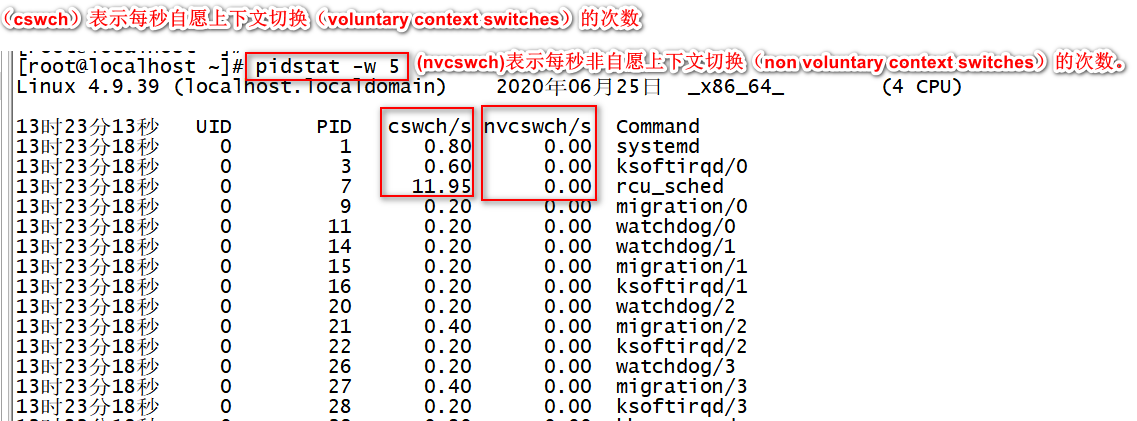
中断上下文切换



# 每隔1秒输出1组数据（需要 Ctrl+C 才结束）# -w参数表示输出进程切换指标，而-u参数则表示输出CPU使用指标

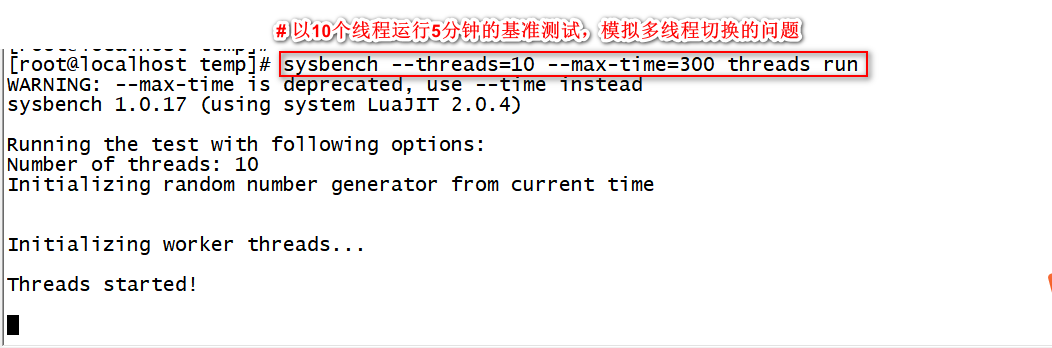
$ pidstat -w -u 1

# 每隔1秒输出一组数据（需要 Ctrl+C 才结束）# -wt 参数表示输出线程的上下文切换指标$ pidstat -wt 1



# 以10个线程运行5分钟的基准测试，模拟多线程切换的问题

$ sysbench --threads=10 --max-time=300 threads run



## 7.36 动态添加系统调用

LinuxTestCode/module/34\_syscall\_ok

如何查找 系统调用首地址

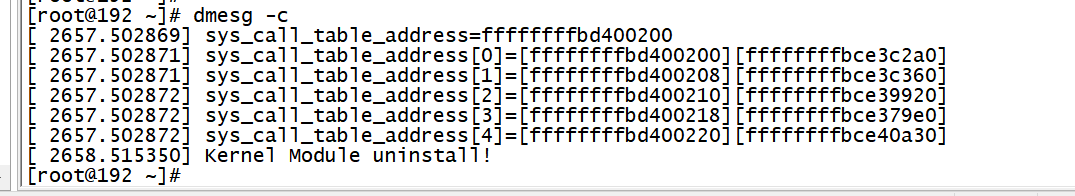
cat /proc/kallsyms | grep sys\_call\_table

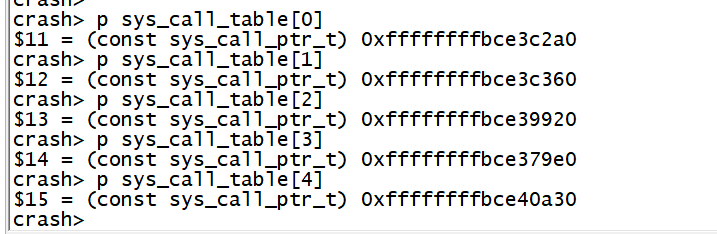
void \*\*syscall\_table;

syscall\_table = (void \*\*) kallsyms\_lookup\_name("sys\_call\_table");

## 7.36 打印系统调用表

LinuxTestCode/module/35\_get\_sys\_call\_table



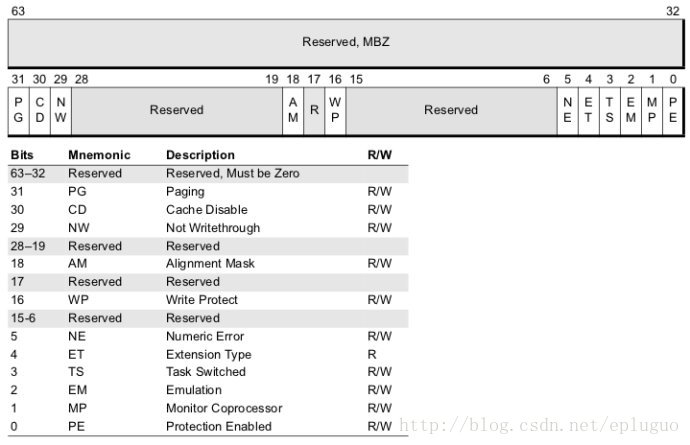


## 7.36 cr0 cr1 cr2 cr3寄存器的理解

LinuxTestCode/module/36\_cr0\_cr1\_cr2\_cr3

cr0 可以控制CPU的一些特性

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| cr0 | | | | | | | | |
|  |  | 读写保护 |  |  |  |  |  |  |
| 31 | ...... | 16 | ...... | .... | 4 | ...... | 1 | 0 |

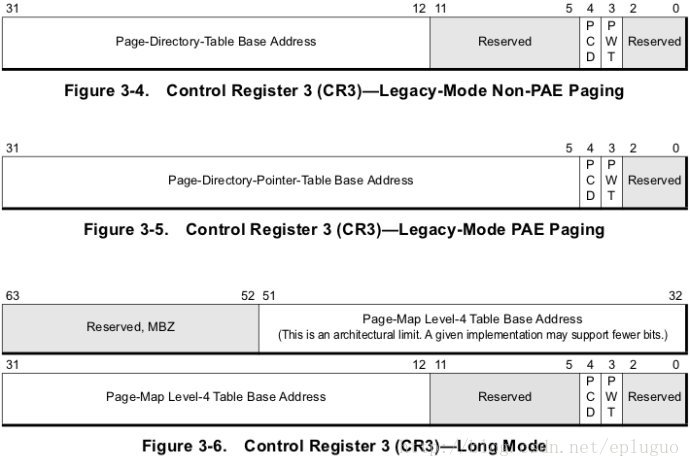


0位是保护允许位PE(Protedted Enable)，用于启动保护模式，如果PE位置1，则保护模式启动，如果PE=0，则在实模式下运行。  
     1 位是监控协处理位MP(Moniter coprocessor)，它与第3位一起决定：当TS=1时操作码WAIT是否产生一个“协处理器不能使用”的出错信号。第3位是任务转换位(Task Switch)，当一个任务转换完成之后，自动将它置1。随着TS=1，就不能使用协处理器。  
     CR0的第2位是模拟协处理器位 EM (Emulate coprocessor)，如果EM=1，则不能使用协处理器，如果EM=0，则允许使用协处理器。  
     第4位是微处理器的扩展类型位 ET(Processor Extension Type)，其内保存着处理器扩展类型的信息，如果ET=0，则标识系统使用的是287协处理器，如果 ET=1，则表示系统使用的是387浮点协处理器。  
     CR0的第31位是分页允许位(Paging Enable)，它表示芯片上的分页部件是否允许工作。  
     CR0的第16位是写保护未即WP位(486系列之后)，只要将这一位置0就可以禁用写保护，置1则可将其恢复

|  |
| --- |
| cr2 |
| CR2是页故障线性地址寄存器，保存最后一次出现页故障的全32位线性地址。 |
| 在发生缺页中断的时候，会 |



|  |
| --- |
| cr3 |
| CR3是页目录基址寄存器，保存页目录表的物理地址，页目录表总是放在以4K字节为单位的存储器边界上，因此，它的地址的低12位总为0，不起作用，即使写上内容，也不会被理会 |
| Cr3寄存器可以求出物理地址 |



## 7.36 slab着色的一些理解

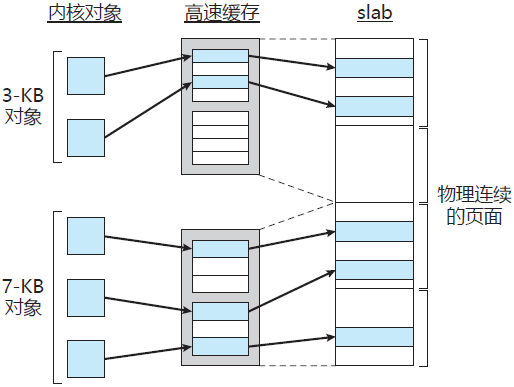
进化流程 slab->slub

CPU读取数据来自cache Line(cache从主存获取的数据)。

cache的数据来自内存的映射如下图所示：

cache是硬件缓存的是内存的数据(部分)，会不会出现数据不一致的问题？

slab管理的是内存（page管理）



着色作用举例子：

cache是cache line 64B 共512行一共（32K），内存大小4G。

如何使用32K的cache映射4G的内存大小，如下图(组相连)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 第0行 | 0-63b | 32k+64B | 64K+64B |  |  |
| 第1行 | 64-127b |  |  |  |  |
| 第2行 |  |  |  |  |  |
| … | … |  |  |  |  |
|  |  |  |  |  |  |
| 511 | 32k | 64k | 96k |  |  |

CPU对cache line读取非常快（只要不跨越cache line）

假如CPU要对0x10000008地址读写，恰巧此时中断来需要读取0x10008008地址的内容。

0x8000 = 32K

0x10000008和0x10008008 对应的cache Line 是同一个位置。

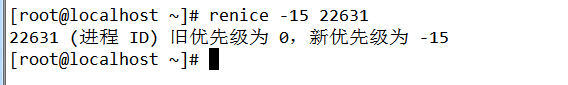
CPU此时检测到有冲突，将cache中0x10000008数据写回物理内存（为啥也写回，不是从内存中加载的不可以清除吗？），然后将0x10008008数据写到cache中，。

假如这样操作1000次，CPU把数据从cache写回内存，然后再把另一块数据写到cache,执行完毕之后，回退操作，比较浪费CPU耗时。

如果能把0x10008008放在cache下一行,这样不是节约时间吗？

此时需要SLAB着色操作：

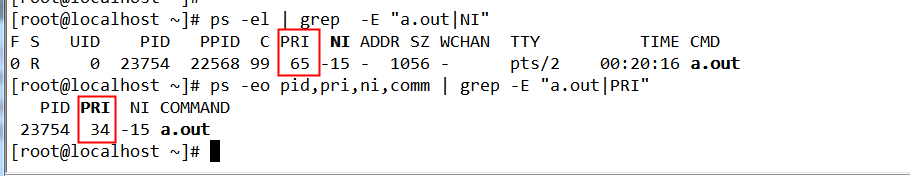
## 7.36 修改nice值使用renice命令



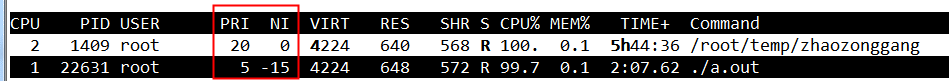
内核优先级：



ps\_PRI = static\_priority-40



top\_PR = static\_Priority-100



## 7.36 poll函数使用说明

https://blog.csdn.net/weixin\_43825537/article/details/90211331

LinuxTestCode/user/16\_poll\_socket

一、在谈epoll和poll网络编程代码之前，首先简单的介绍一下poll函数和epoll函数组

1、poll函数原型

使用此函数的头文件为

#include<poll.h>

poll函数原型如下：

#include<poll.h>

int poll(struct poll\_fd \*fds, nfds\_t nfds, int timeout)

参数解释

（1）fds：指向元素类型为struct poll\_fd类型的首元素

（2）nfds：fds结构体数组的大小

（3）timeout：表示poll函数超时的时间限制

poll\_fd结构

struct poll\_fd

{

int events;

int revents;

int fd;

}

变量说明

（1）events:告诉内核需要检测的事件（此为用户设置）

（2）revents：对文件描述符进行操作结果事件，在内核中进行操作（内核返回时的完成时间）

（3）fd：每个结构体中都会有一个被监视的文件描述符，内核可以处理多个结构体，就说明可以监测多个文件描述符

以下为events和revents的取值



2、poll函数实现的原理

内核将用户的fds结构体数组拷贝到内核中，当有时间发生时内核再将所有时间都返回到fds数组中，

polll函数只返回已就绪时间的个数，所以用户要操作就绪事件，就得用轮询的方法

二、poll网络编程实例

注意：poll函数是阻塞函数，当没有就绪文件描述符的时候，poll一直处于阻塞状态，知道有就绪文件描述符

程序代码（包含注释）

三、poll的优缺点

1、优点

（1）poll() 不要求开发者计算最大文件描述符加一的大小。

（2）poll() 在应付大数目的文件描述符的时候速度更快，相比于select。

（3）它没有最大连接数的限制，原因是它是基于链表来存储的。

（4）在调用函数时，只需要对参数进行一次设置就好了

2、缺点

（1）大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义（epoll可以解决此问题）

（2）与select一样，poll返回后，需要轮询pollfd来获取就绪的描述符，这样会使性能下降

（3）同时连接的大量客户端在一时刻可能只有很少的就绪状态，因此随着监视的描述符数量的增长，其效率也会线性下降

## 7.36 poll函数使用说明

LinuxTestCode/user/15\_my\_socket

自己写的UDP协议封装类型

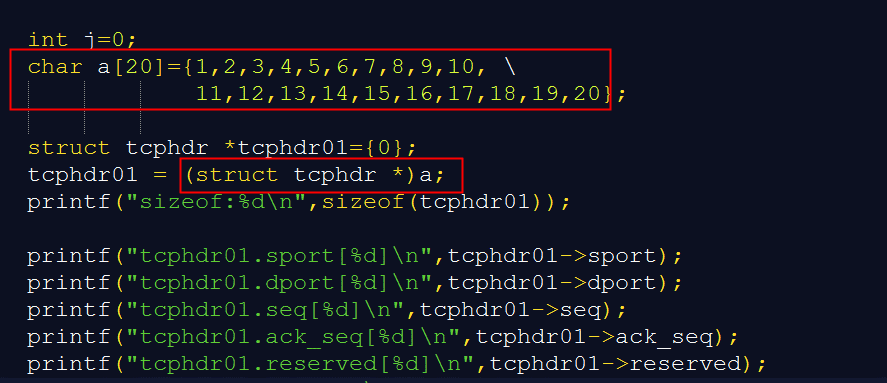
（见表格分析）

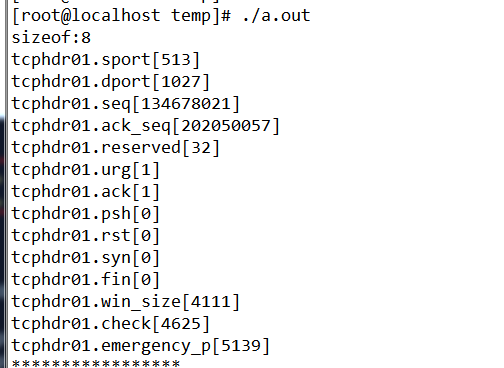
## 7.36 自己添加tcp协议头

LinuxTestCode/user/18\_my\_tcphdr

将一串数据强制转换（struct tcphadr \*）然后分析结果

（见表格分析）





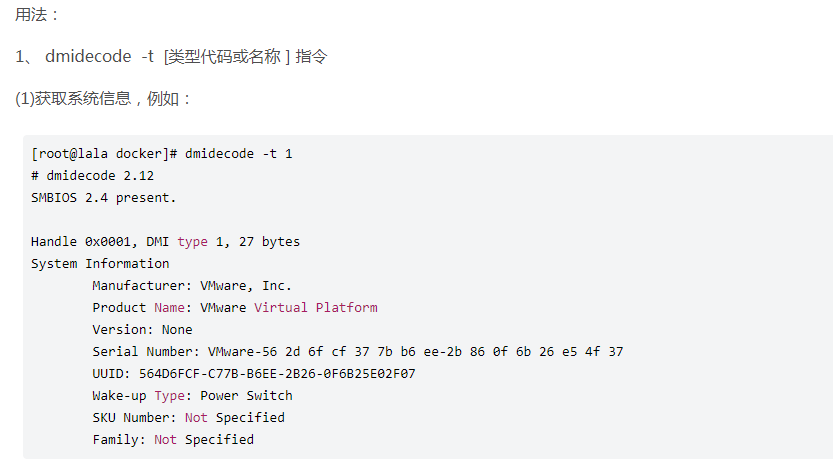
## 7.36 TLB和cache关系

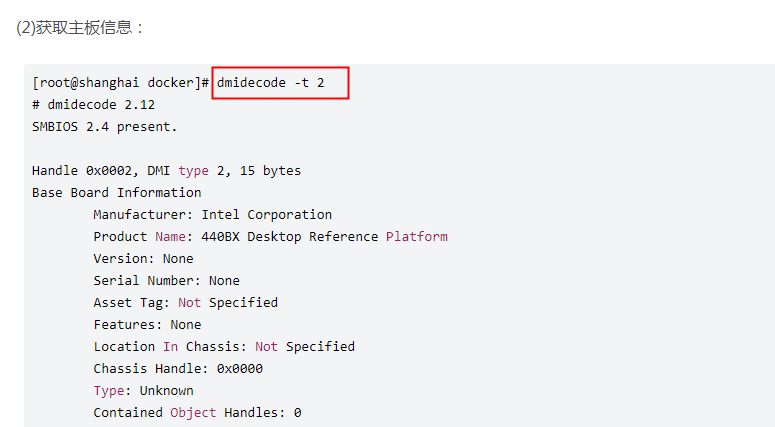
按照网上的说法：

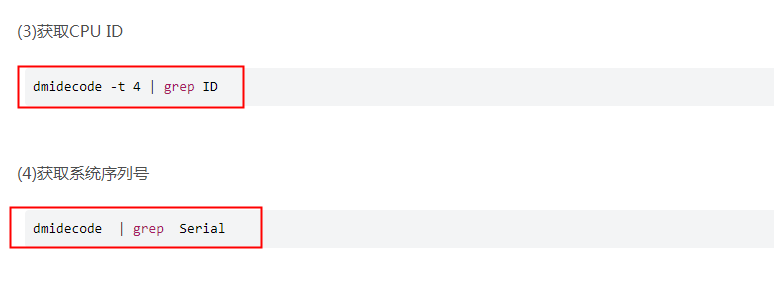
TLB是个缓存器（虚拟到物理地址的映射）

cache也是一个高速缓存（CPU访问主存慢）

## 7.36 linux dmidecode 获取CPU ID等系统硬件信息









## 7.36 个人对PTE的理解

pgd = pgd\_offset(current->mm,vaddr);

p4d = p4d\_offset(pgd, vaddr);

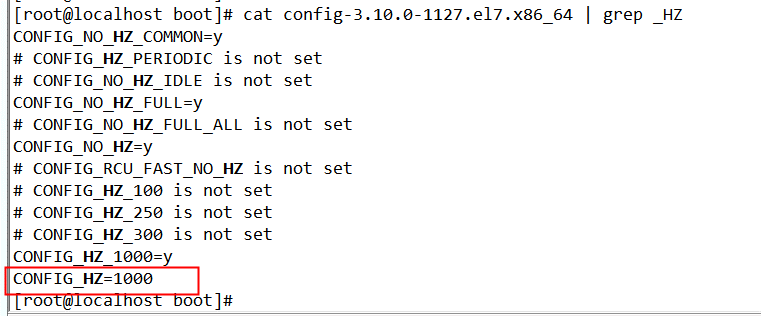
pud = pud\_offset(p4d, vaddr);

pmd = pmd\_offset(pud, vaddr);

pte = pte\_offset\_kernel(pmd, vaddr);

pte的求法和其他的不太一致，原因是否是因为PTE存在查表格得出

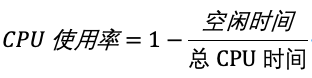
## 7.36 进程在CPU占有100%如何排查原因



Jiffies记录开机以来的节拍数字

此系统下1s产生1000次中断（这个属于内核空间）

用户空间设置了100，即1s 100次中断

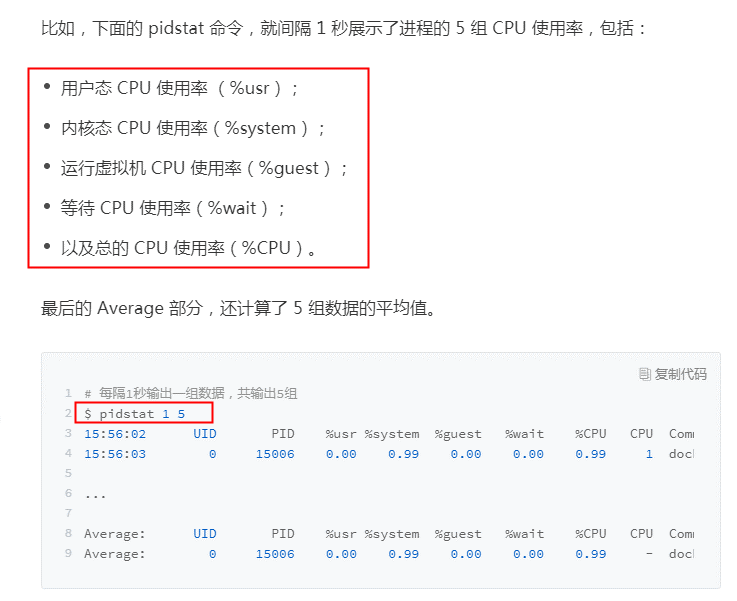


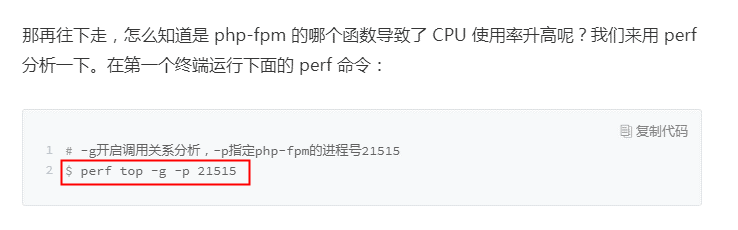
这个计算方法参考不大：

cpu利用率计算最近的数值参考比较好：



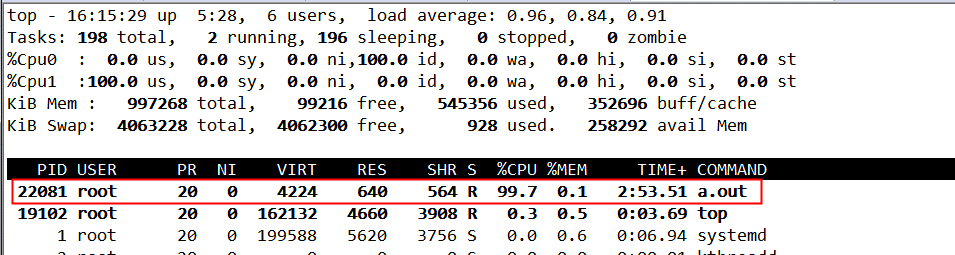
如何查看进程CPU使用率？



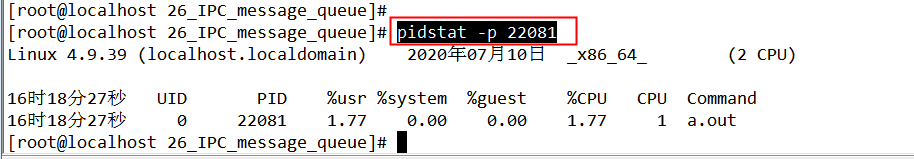


总结流程

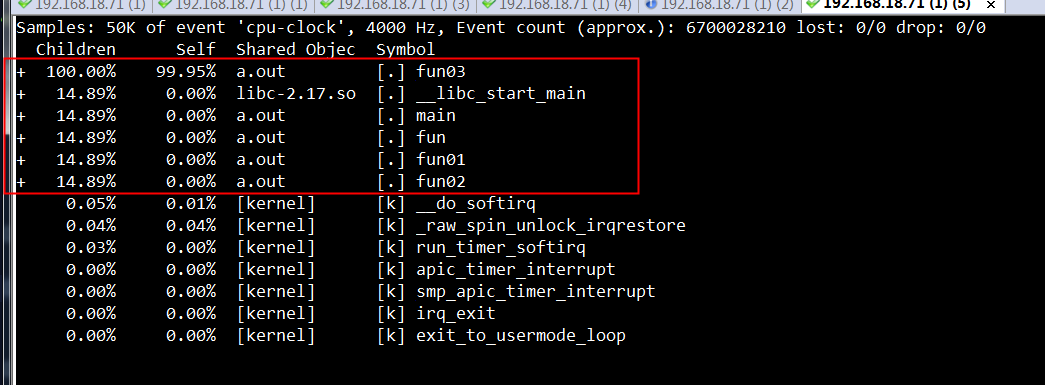
【01】使用top 查看那个进程CPU使用率高



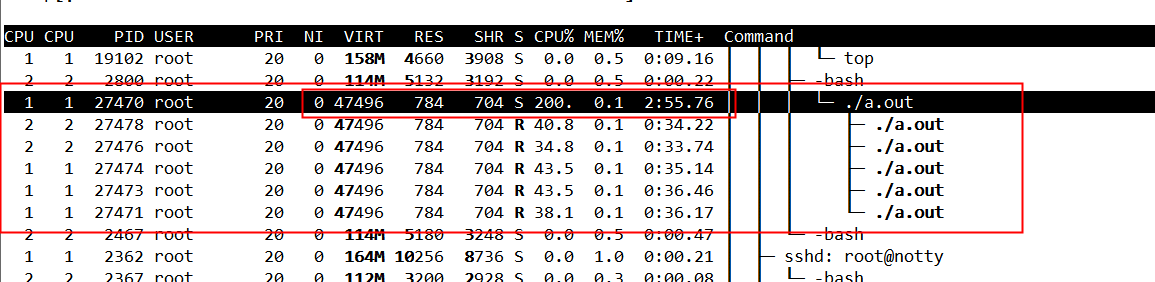
【02】使用pidstat -p 22081 查看进程占用CPU高的是那个 空间

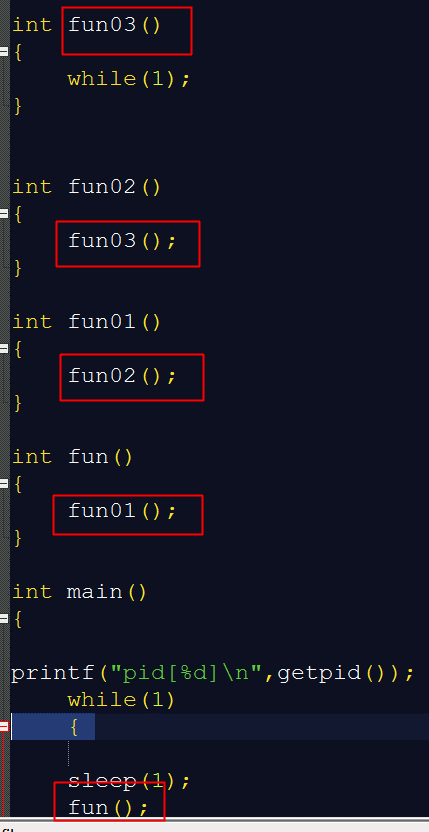


【03】perf top -g -p 22081 查看那个函数占用了CPU



进程占用200%的CPU，每个线程可以在不同核上运行。





## 7.36 发生了死锁怎么办？

我感觉无解，

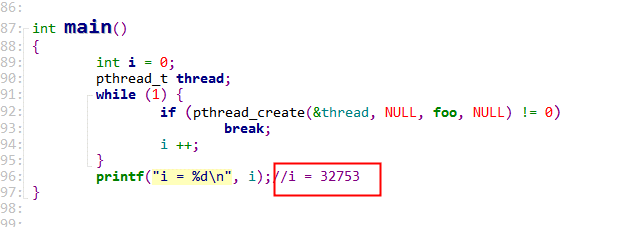
只能查处死锁的进程，死锁的代码，修改代码。

## 7.36 如何处理僵尸进程？

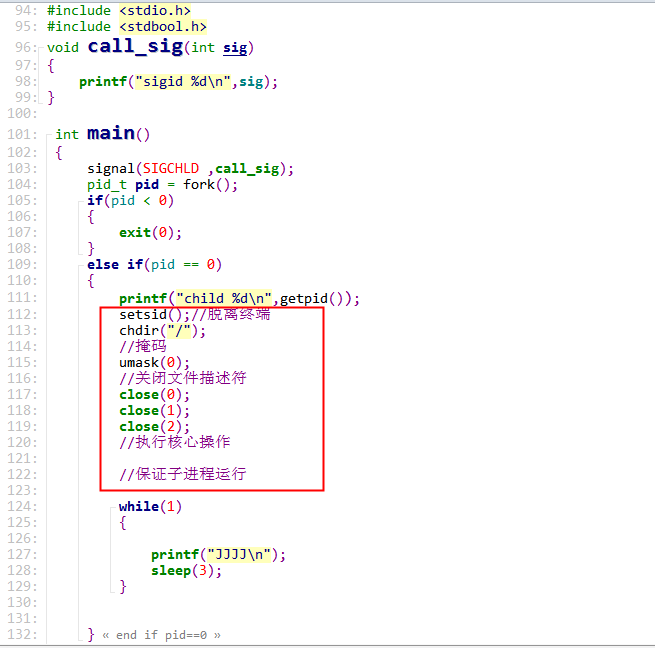
方法一：杀死父进程 托管给 init进程，让init进程处理僵死进程

方法二：kill –HUP (PID) 发送挂起信号，好像不能用

## 7.36 Linux下一个进程可以开多少线程



## 7.36 守护进程的创建流程



## 7.36 获取CPU权限的操作

1 内核抢占（就是强制切换进程CPU）

preempt\_enable()  开启抢占  
preempt\_disable() 禁止抢占

2 中断

## 7.36 获取CPU权限的操作

## 7.36为什么自旋锁不能睡眠 而在拥有信号量时就可以

按照原理来说是：自旋锁是关闭抢占的，死循环，独占CPU

信号量是可以抢占。

但是问题是？信号量的实现靠自旋锁，为什么信号量代码还可以调度。

## 7.36硬件中断是如何响应



当外部产生中断之后，交给GPIO控制器，交给GIC控制器，交给CPU响应。

当CPU接收到电信号之后，反推流程：

CPU获得中断信号，需要查询GIC（寄存器）中断号（A发生中断，A也有自己的中断描述符）。

第一：irq\_desc[A]. handle\_irq回调函数查找GPIO中断控制器（寄存器）那个中断发生了（B发生中断，B也有自己的中断描述符）。

第二：由第一步：查询线到了B中断的发生，调用irq\_desc[B]. handle\_irq调用外部设备1的处理函数（irq\_desc[B]. handle\_irq 操作irqaction 结构体中的函数）。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

当硬件中断号小于16时中断为软件产生的中断（SGI），中断处理程序调用handle\_IPI（）函数处理；当硬件中断号大于或等于16而小于32时中断为私有外设中断（PPI），handle\_irq（）函数被设置为handle\_percpu\_devid\_irq（）；当硬件中断号大于等于32但小于gic\_data.irq\_nr（该值在irq-gic-v3.c文件中的gic\_init\_bases（）函数中初始化，其最大值为1020，表示GICv3最多支持1020个中断源（SGI+PPI+SPI））时中断为共享外设中断，handle\_irq（）被设置为handle\_fasteoi\_irq（）；当硬件中断号大于等于8192但小于GIC\_ID\_NR时中断为局部特定外设中断（LPI），handle\_irq（）被设置为handle\_fasteoi\_irq（）。有关中断类型的知识可以在第二十八期找到。设置handle\_irq（）的代码可以在openEuler源码仓库

## 7.36 内核操作文件方法

LinuxTestCode/module/51\_kernel\_read\_write\_file

# 第8章 linux开源项目

## 8.1 设计一个需求

LinuxTestCode/open\_source\_project/01\_print\_filesystem\_kernel\_parameter

1 cat /proc/allparameter\_d xxx

pid:xxx

task\_struct:xxx

mm\_struct:xxx

vm\_area\_struct:xxx

maj\_flt:xxx

min\_flt:xxx

page\_file:xxx

page\_anno:xxx

page\_ swap:xxx

//自己计算某一进程的（多核）CPU使用率和top命令对比

//总的 CPU使用率和top命令对比

## 8.1 设计需求动态修改系统调用

/LinuxTestCode/module/34\_syscall\_ok

# 第9章 linux内核面试题

地址：https://blog.csdn.net/laviolette/article/details/51481175

## 9.1 Linux中主要有哪几种内核锁

【1】自旋锁和信号量

## 9.1 Linux用户和内核模式什么意思

## 9.1 Linux怎样申请大块内存

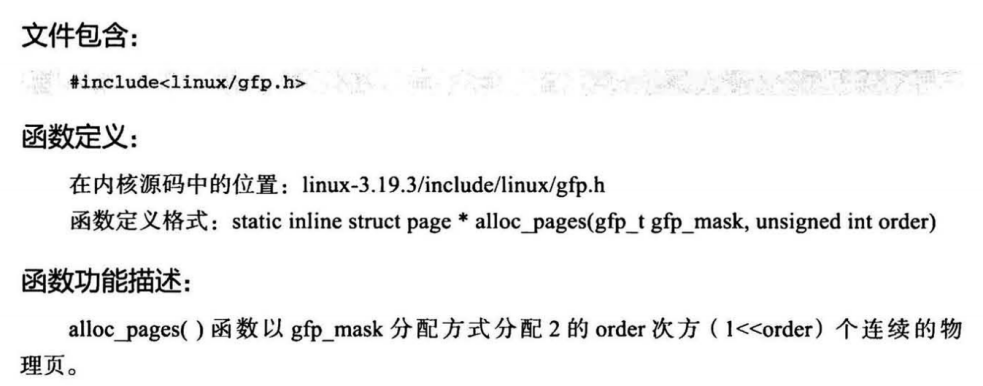
Vmalloc申请不连续的页面空间

## 9.1 用户进程间通信的方法

【管道】【匿名管道】【消息队列】【信号量】【共享内存】【套接字】

## 9.1 通过伙伴系统申请内核的函数有哪些





alloc\_pages(gfp\_mask, order)，

\_ \_get\_free\_pages(gfp\_mask, order)

## 9.1 通过slab分配器申请内核函数

Kmem\_cache\_create()//创建slab

Kmem\_cache\_alloc()//申请一个对象

//include/linux/slab.h # 基于slab分配的内存

Kmalloc()->Kmem\_cache\_alloc()//kmalloc()最大申请128K大小 对应物理内存

//分配虚拟地址

Vmalloc()

## 9.1 Linux的内核空间和用户空间是如何划分的以32位系统为例）？

1-3 3-4

## 9.1 vmalloc申请内存有何特点？

虚拟连续

## 9.1 用户程序使用malloc()申请到的内存空间在什么范围？

用户堆区

## 9.1 在支持并使能MMU的系统中，Linux内核和用户程序分别运行在物理地址模式还是虚拟地址模式？

Linux内核和虚拟都运行在虚拟地址模式

对于没有mmu系统中，不区分内核和用户空间

Linux针对有mmu所设计的系统，虚拟地址送至mmu,mmu把虚拟地址映射为物理空间，通过赋予每个任务不同的虚拟-物理地址转换映射，支持不同任务之间的映射。

## 9.1ARM处理器是通过几级也表进行存储空间映射的？

Arm32->3

Arm64->4

## 9.1 Linux虚拟文件系统的关键数据结构有哪些？至少写出四个）

## 9.1 创建进程的系统调用有那些？之间区别和联系

<https://blog.csdn.net/gogokongyin/article/details/51178257>

long [**do\_fork**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/do_fork)(unsigned long [**clone\_flags**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/clone_flags),//创建进程标志位集合

unsigned long [**stack\_start**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/stack_start),//栈开始地址

unsigned long [**stack\_size**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/stack_size),//栈大小

int [**\_\_user**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/__user) \*parent\_tidptr,//用户指针指向父进程PID

int [**\_\_user**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/__user) \*child\_tidptr) //用户指针指向子进程PID

{

return [**\_do\_fork**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/_do_fork)([**clone\_flags**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/clone_flags), [**stack\_start**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/stack_start), [**stack\_size**](https://elixir.bootlin.com/linux/v4.9.39/C/ident/stack_size),

parent\_tidptr, child\_tidptr, 0);

}

fork()->sys\_clone->do\_fork()

do\_fork(SIGCHLD , 0, 0,NULL, NULL)

｛

struct task\_struct \*p;

p = copy\_process(clone\_flags, stack\_start, stack\_size,

child\_tidptr, NULL, trace, tls, NUMA\_NO\_NODE);

｝

Clone()->sys\_clone()->do\_fork()

do\_fork(clone\_flag , newsp , 0, parent\_tidptr, child\_tidptr)

例子：pthread\_create()

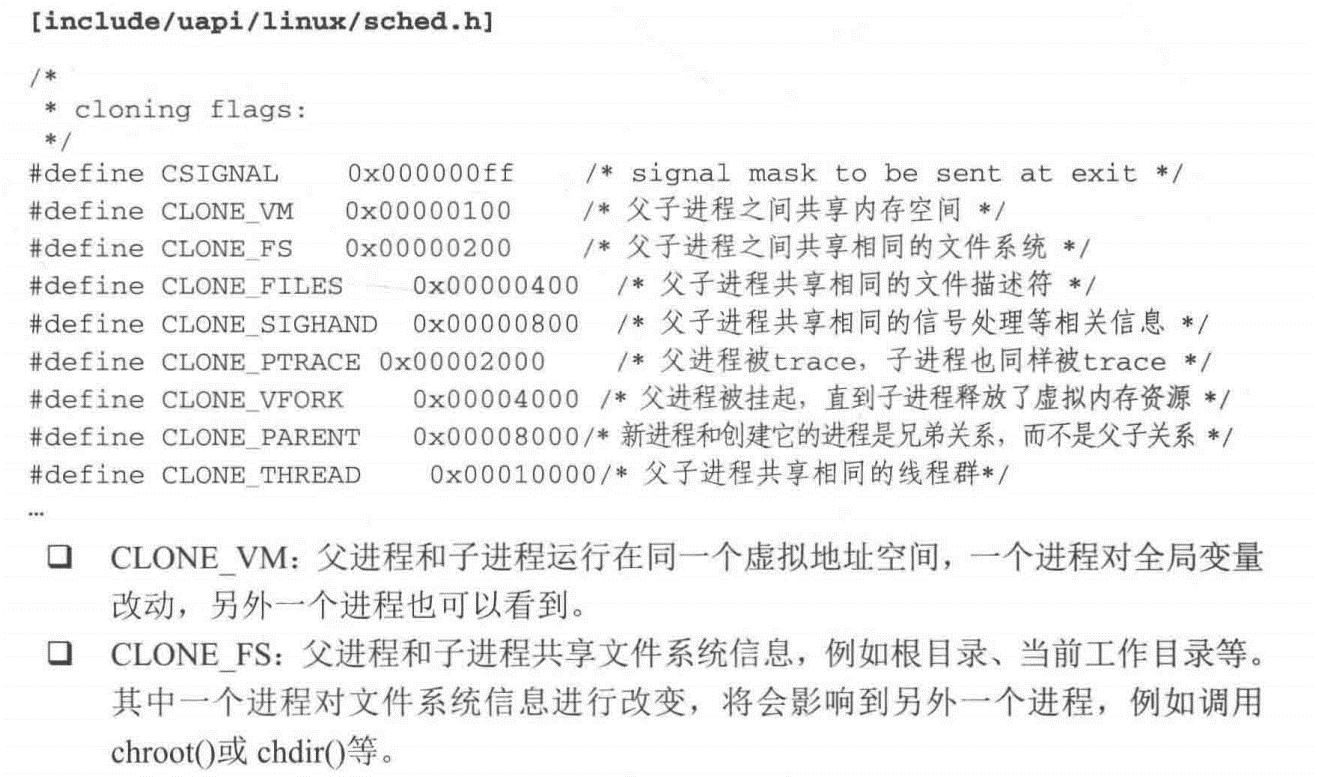
//vfork过时了

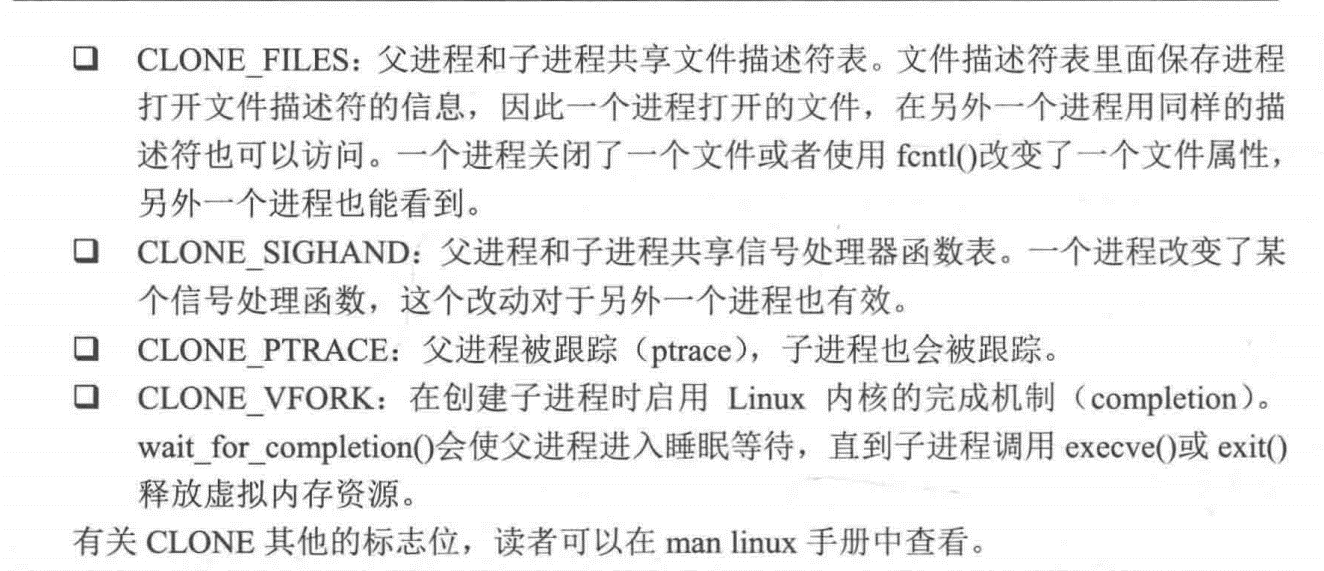
Vfork()->sys\_vfork()->do\_fork()

do\_fork(CLONE\_VFORK | CLONE\_VM | SIGCHLD , 0, 0,NULL, NULL)

内核线程：

do\_fork(flags | CLONE\_VM | CLONE\_UNTRACED , (unsigned long)fn,(unsigned long)arg 0,NULL, NULL)





## 9.1 调用schedule()进行进程切换的方式有几种

<https://www.doc88.com/p-6803246858710.html>

系统调用do\_fork()

定时中断do\_timer()

唤醒进程 wake\_up\_process()

改变进程的调度策略setscheduler()

系统调用礼让sys\_sched\_yield()

## 9.1 Linux中的浮点运算由应用程序实现还是内核实现

应用程序的实现是使用库函数。

## 9.1 TLB缓存的内容是什么

属于页表缓存，当第一次 线性地址转换物理地址，将线性地址物理地址存放TLB表中。

## 9.1 如何避免死锁

加锁按照一定的顺序，避免死锁

加锁要有一定时限，超时，自动释放自己已有的锁

避免一个线程获得多个锁

避免一个锁获的多个资源

## 9.1 netstat tcpdump ipcs ipcrm 命令的使用

地址：https://www.runoob.com/linux/linux-comm-netstat.html

netstat 由于显示网络状态

地址：https://www.runoob.com/linux/linux-comm-tcpdump.html

tcpdump 解析网络数据

地址：https://blog.csdn.net/dalongyes/article/details/50616162

ipcs -a命令可以查看当前使用的共享内存、消息队列及信号量所有信息，对于该选项对应的结果

地址：<https://www.cnblogs.com/MartinChentf/p/6057100.html>

ipcrm通过指定ID删除删除IPC资源，同时将与IPC对象关联的数据一并删除，只有超级用户或IPC资源创建者能够删除

## 9.1 系统如何把信号通知到进程？

比如kill -9 <pid> 如何杀死进程的？

信号分类：

* 1~31之间的信号叫做非实时信号, 不支持排队, 信号可能会丢失, 也叫做不可靠信号。
* 34~64之间的信号叫做实时信号, 支持排队, 信号不会丢失, 也叫做可靠信号。

可靠信号注册机制:

内核每收到一个可靠信号都会去注册这个信号，在信号的未决信号链中分配sigqueue结构，因此，不会存在信号丢失的问题。

* 不可靠信号的注册机制:

而对于不可靠的信号，如果内核已经注册了这个信号，那么便不会再去注册，对于进程来说，便不会知道本次信号的发生。

信号产生->信号注册－>信号在进程中注销->信号处理函数执行完毕

kill()系统调用（sys\_kill）进入内核, 内核根据传进来的进程和sig,首先判断是否有权限，

如果有权限，系统设置目标进程的PCB（bitmap）,设置完成内核->目标进程发送，中断请求。

/\*\*\*\*\*\*/

exit\_to\_usermode\_loop(struct pt\_regs \*regs, u32 cached\_flags)

do\_signal(regs);

handle\_signal(&ksig, regs);// 信号处理就是调用用户提供的信号处理函数

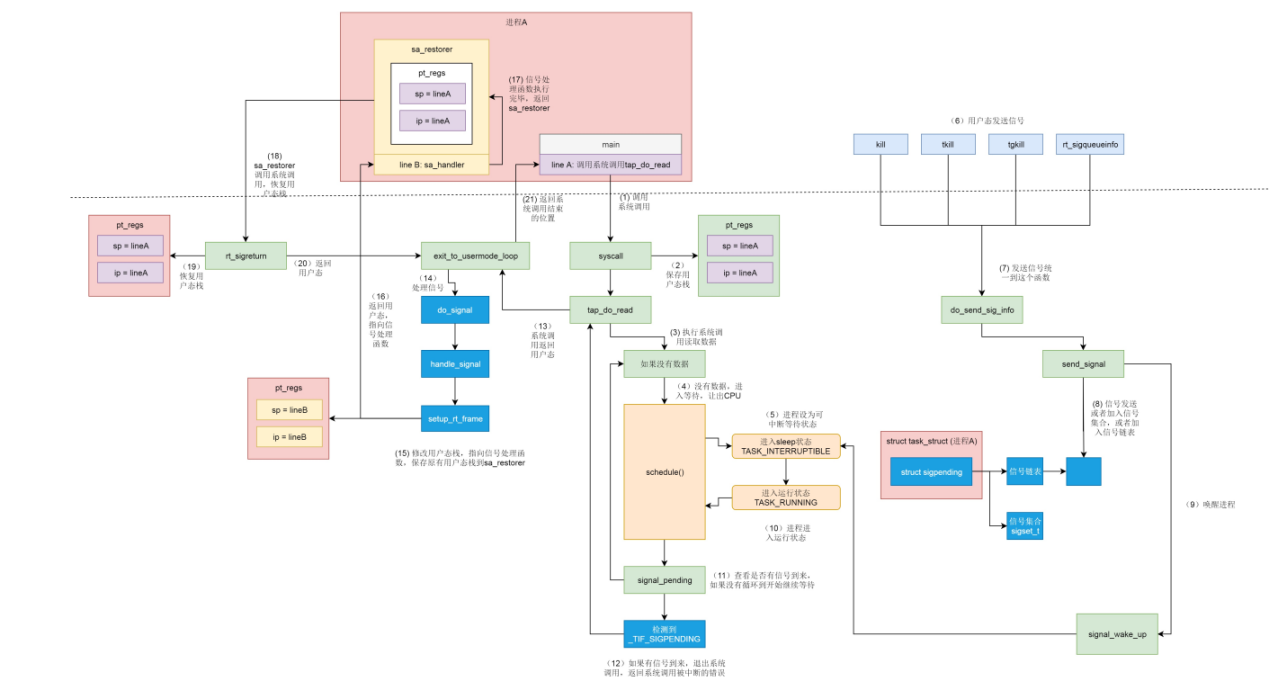
case -ERESTARTSYS:  
if (!(ksig->ka.sa.sa\_flags & SA\_RESTART)) {  
regs->ax = -EINTR;  
break;

/\*\*\*\*\*\*/

setup\_rt\_frame()//建立用户态堆栈

当进程又切换到用户态时，因为信号处理程序的起始地址被强制放进程序计数器中，因此开始执行信号处理程序

setup\_frame()函数把一个叫做帧(frame)的数据结构推进用户态堆栈中，这个帧含有处理信号所需要的信息，并确保正确返回到handle\_signal()函数



## 9.1 mmap实现原理

匿名映射 malloc

文件映射

映射包括：

虚拟到物理的映射：

虚拟到文件的映射：