# 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>

*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

## 面试总结:

# 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

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 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

*/\**

*\* 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 \*/*

unsigned long [**address**](https://elixir.bootlin.com/linux/v5.0.21/ident/address); */\* Faulting virtual address \*/*

[**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

### 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

# 第四章锁与进程间通信

## 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

#### 4.4.4.1 管道

#### 4.4.4.1 套接字

就是socket

## 4.1页面机制