深入理解linux内核读书笔记

# 第三章 进程

1、创建进程

目前三种机制：写时复制，轻量级进程共享进程的内核数据结构，vfork创建进程共享其父进程的内存地址空间。

（1）clone，fork，vfork

轻量级进程(即线程)pthread\_create会调用clone创建，clone()对应的系统调用是sys\_clone()。

fork和vfork都是调用clone创建，但是传递给clone的flags不一样。fork子进程是父进程的副本且写时复制，而vfork共享地址空间同时保证子进程先运行。

三个系统调用对应的内核服务例程分别是：sys\_clone, sys\_fork, sys\_vfork，最终都会调用do\_fork进行操作。另外创建内核线程kernel\_thread也是调用do\_fork进行的。

（2）do\_fork

参数regs：指向通用寄存器的指针，通用寄存器值从用户态切换到内核态时被保存到内核态堆栈中的。

do\_fork使用copy\_process来创建进程描述符以及子进程执行所需要的所有其他内核结构数据。

（3）内核线程

通过kernel\_thread创建内核线程，只运行在内核态使用大于PAGE\_OFFSET的线性地址空间。

进程0（idle进程）linux初始化阶段从无到有的一个内核线程，使用静态分布的数据结构（其他进程都是动态分配）：

init\_task：存放进程描述符

init\_thread\_union：存放thread\_info描述符和内核堆栈。

进程描述符指向列表：init\_mm，init\_fs，init\_files，init\_signals，init\_sighand。

主内核页全局目录存放在swapper\_pg\_dir中。

进程0在start\_kernel()函数初始化内核需要的所有数据结构。其中sched\_init()完成调度器初始化和建立进程0，仅仅是为调度做准备；rest\_init()里面创建了进程1和进程2，分别是：kernel\_init和kthreadd，进程1与进程0共享进程的所有内核数据结构；然后进程0执行cpu\_idle函数。

进程1调度后，将通过kernel\_execve()系统调用装入可执行程序init。该init就是busybox编译出来的init程序，启动/etc下面的脚本，然后运行各个进程。

常见内核线程：

Keventd：执行keventd\_wq工作队列中的函数

Kapmd：处理与高级电源管理（APM）相关的事件

Kswapd：执行内存回收

Pdflush：刷新“脏”缓冲区中的内容到磁盘以回收内存

Kblockd：执行kblockd\_workqueue工作队列中的函数。实质是周期性激活块设备驱动程序

Ksoftirqd：运行tasklet

Kworker: 工作者线程类似kblockd用于运行工作队列（2.4前叫任务队列）

# 第四章 中断

中断分为同步和异步。同步（也称异常）由CPU控制单元产生；异步（也称中断）由间隔定时器和I/O设备中断产生。

异常（同步）：一种由程序的错误产生，通过信号处理异常；另一种由内核必须处理的异常条件产生，通过内核执行恢复异常需要的所有步骤完成，如缺页请求、系统调用内核服务请求。

中断处理时内核执行最敏感任务之一：立即响应，处理紧急，延迟非紧急；中断嵌套处理；临界区中断必须禁止。

1、初始化中断描述符表

这项工作在初始化系统时完成。start\_kernel函数里面调用：trap\_init、early\_irq\_init、init\_IRQ、softirq\_init。

2、中断处理程序

由三部分组成：

（1）在内核堆栈中保存大多数寄存器的内容

（2）用高级的C函数处理中断

（3）通过ret\_from\_exception函数从中断处理程序退出

3、中断处理

I/O中断，时钟间隔中断，处理器间中断。

（1）I/O中断处理

两种方式实现：IRQ共享，IRQ动态分配。early\_irq\_init函数初始化IRQ中断，每个中断向量有一个irq\_desc描述符，一个处理器的所有IRQ组成一个中断向量数组。描述符数据结构irq\_desc中的action就是该IRQ对应的中断服务例程列表（如果未共享只有一个，共享则有多个）。

（2）时钟间隔中断

由init\_IRQ函数初始化本地timer定时器

（3）中断统一处理入口asm\_do\_IRQ

asm\_do\_IRQ函数调用generic\_handle\_irq函数处理每个中断描述符对应的中断服务例程。

4、IRQ线动态分配

有几个中断向量留给特定的设备，其余的向量都是被动态处理。通过request\_irq函数来注册中断，该函数调用request\_threaded\_irq函数，接着调用\_\_setup\_irq来中断注册，放到action队列中。

注：以上可以认为是硬中断的处理。

5、软中断（也叫中断后半部处理）

内核支持一下几种软中断：

enum

{

HI\_SOFTIRQ=0, //处理优先级的tasklet

TIMER\_SOFTIRQ, //和定时器中断相关的tasklet

NET\_TX\_SOFTIRQ, //把数据包传送给网卡

NET\_RX\_SOFTIRQ, //从网卡接收数据包

BLOCK\_SOFTIRQ, //

BLOCK\_IOPOLL\_SOFTIRQ, //

TASKLET\_SOFTIRQ, //处理常规tasklet

SCHED\_SOFTIRQ, //

HRTIMER\_SOFTIRQ, //

RCU\_SOFTIRQ, /\* Preferable RCU should always be the last softirq \*/

NR\_SOFTIRQS

};

两种非紧迫和可中断的内核函数：可延迟函数（软中断和tasklets）和通过工作队列来执行的函数。

6、软中断和tasklets

（1）软中断和tasklet有密切关系，tasklet是在软中断之上实现的。软中断可以并发运行在多个CPU上，软冲断是可重入函数，且必须明确使用自悬锁保护其数据结构。Tasklet不能再两个CPU上同时运行相同类型的tasklet，不必是可重入的。

（2）一般有四个操作：初始化（内核初始化或驱动加载）、激活（任何时刻，一般硬中断处理里面）、屏蔽（可以不执行）、执行（执行同类型的所有挂起延迟函数）。

（3）软中断的数据结构：struct softirq\_action softirq\_vec[NR\_SOFTIRQS]， softirq\_vec数组每一个元素通过下标对应上面枚举里每种软中断，每个软中断有一个action软中断处理函数。例如TASKLET\_SOFTIRQ对应的action是tasklet\_action，该函数被调用 会扫描处理所有tasklet。

raise\_softirq函数用来激活软中断，接受中断下标nr作为参数。do\_softirq函数则扫描softirq\_vec数组，执行对应的action，action处理对应软中断的元素，例如上面的tasklet\_action。

因此，需要周期性的（不能太频繁）检查活动（挂起）的软中断，以下几个会触发do\_softirq函数（不同内核版本情况不一样）：

A．当内核调用local\_bh\_enable函数激活本地CPU的软中断

B．当asm\_do\_IRQ完成I/O中断处理后调用irq\_exit()宏

C．在多处理器系统中，当CPU处理完被CALL\_FUNCTION\_VECTOR处理器间中断所触发的函数。

D．当一个特殊的ksoftirqd/n内核线程被唤醒

（4）tasklet

tasklet是I/O驱动程序中实现可延迟函数的首选方法。

基本工作流程：

初始化：申请一个tasklet\_struct数据结构，调用tasklet\_init

激活tasklet：根据优先级调用tasklet\_schdeule或tasklet\_hi\_schedule，最终调用raise\_softirq\_irqoff或raise\_softirq激活TASKLET\_SOFTIRQ或HI\_SOFTIRQ类型的软中断。

运行tasklet：软中断激活后就会执行do\_softirq函数，然后调用tasklet对应的action。

7、工作队列

2.6引入工作队列，替换2.4的任务队列。允许内核函数被激活，由工作者线程（worker thread）的特殊内核线程执行。

与上面的主要区别：上面一般用于中断上下文，工作队列函数运行在进程上下文；但都不能访问进程的用户空间。

具体用法和区别有待进一步深入。

# 第五章 内核同步

内核看作是不断对请求进行响应的服务器，请求来自在CPU上执行的进程，可能来自发出中断请求的外部设备。

1、内核为不同服务提供请求

内核看作必须满足两种请求的侍者：一种请求来自顾客（系统调用或异常），一种来自数量有限的老板（中断）。侍者提供的服务对应CPU处于内核态时所执行的代码，CPU在用户态执行，则侍者认为处于空闲状态。不同请求，侍者采用如下策略：

（1）老板提出请求时，如果侍者正空闲，则为老板服务

（2）老板提出请求时，侍者正为顾客服务，则停止为顾客服务，开始为老板服务。

（3）老板提出请求时，侍者正为另一个老板服务，则停止第一个老板服务，开始为第二个老板服务，服务完毕再为第一个老板服务

（4）一个老板可能命令侍者停止正在为顾客提供的服务，侍者完成对老板最近请求服务后，可能暂时不处理原来的顾客而去为新选中的顾客服务。结合（2）点。

2、内核抢占

如果进程正执行内核函数时即在内核态运行，运行发生内核切换，这个内核是可以抢占的；主要特点：一个内核态运行的进程，可能在执行内核函数期间被另外一个进程取代。而非抢占内核：内核进程运行直到自动放弃CPU为止。

当被current\_thread\_info()宏所引用的thread\_info描述符的preempt\_count字段大于0时，就禁止内核抢占。只有当内核正在执行异常处理程序(尤其是系统调用)，而且内核抢占没有被显式的禁用时，才可能抢占内核。

处理preempt\_count字段宏如下：

preempt\_count()：获取preempt\_count字段

preempt\_disable()：使抢占计数器值+1

preempt\_enable\_no\_resched()：使抢占计数器值-1

preempt\_enable()：使抢占计数器值-1，并在thread\_info描述符字段TIF\_NEED\_RESCHED标志被置1的情况下，调用preempt\_schedule()。

get\_cpu：与preempt\_disable相似，但返回本地CPU数量

put\_cpu：与preempt\_enable相同

put\_cpu\_no\_resched()：与preempt\_enable\_no\_resched相同

3、内核同步原语

内核控制路径交错执行时避免共享数据之间的竞争条件，linux内核使用的同步技术如下：每CPU变量、原子操作、内存屏障、自旋锁、信号量、顺序锁、本地中断禁止、本地软中断禁止、读拷贝更新(RCU)。

4、每CPU变量

把内核变量声明为每CPU变量，每CPU变量主要是数据结构的数组，每个CPU对应数组的一个元素。内核抢占可能使每CPU变量产生竞争条件，因此内核控制路径应该在禁用抢占的情况下访问每CPU变量。对应的操作宏：DEFINE\_PER\_CPU，per\_cpu，get\_cpu\_var等

5、原子操作

读—修改—写，一次完成。

常见原子操作：atomic\_inc，atomic\_add等，但需要声明为原子类型atomic\_t。原子位处理：test\_bit，set\_bit，clear\_bit等

6、优化屏障和内存屏障

处理同步时，必须避免指令由于编译器优化而重新排序。

优化屏障：不会重新排序。Linux使用宏：barrier()——arm volatile(“”:::”memory”)，但是无法保证不使CPU把汇编语言指令混在一起执行——内存屏障工作。

内存屏障：在原语之后的操作开始执行之前，原语之前的操作已经完成。宏：mb()、rmb()、wmb()。

7、自旋锁

（1）自旋锁

自旋锁用来在多处理器环境中工作的一种特殊的锁。一般来说，由自选锁保护的每个临界区都是禁止内核抢占的。自旋锁操作宏：spin\_lock\_init，spin\_lock，spin\_unlock, spin\_unlock\_wait, spin\_is\_locked, spin\_trylock。

（2）读/写自旋锁

允许多个内核控制路径同时读一个数据结构，但是只能有一个控制路径可以写改数据结构。对应的宏：rwlock\_init，read\_lock，write\_lock。

缺点：读和写具有相同的优先级，读必须等待直到写操作完成，写也必须等待直到读完成。从而引入顺序锁。

（3）顺序锁

类似读/写锁 2.6引入，只是为写者赋予高优先级，即在读的时候允许写继续运行。导致要多次读直到有效为止。对应宏：seqlock\_init，write\_seqlock or write\_sequnlock，read\_seqbegin or read\_seqretry。

8、读-拷贝-更新（RCU）

RCU为了保护在多数情况下被多个CPU读的数据结构而设计的另一种同步技术，允许对个读者和写者并发执行。

使用范围：RCU只保护被动态分配并通过指针引用的数据结构；在被RCU保护的临界区中，任何内核控制路径都不能睡眠。

设计宏：rcu\_read\_lock，rcu\_read\_unlock。

读者间接引用该数据结构指针所对应的内存单元并开始读这个数据结构；写者要更新数据结构时，间接引用指针并生成整个数据结构的副本，修改完成，更改指向数据结构的指针。待所有读者都执行完rcu\_read\_unlock后，释放旧副本call\_rcu。

9、信号量

两种信号量：内核信号量（由内核控制路径使用）和System V IPC信号量（由用户态进程使用）。

（1）信号量

对应的宏：up 和 down or down\_trylock, 读/写信号量：up\_read or up\_write 和 down\_read or down\_write or down\_read\_trylock or down\_write\_trylock。

（2）补充原语

解决多处理器系统上发生的一宗微妙竞争条件。对应的接口：complete 和wait\_for\_completion。

10、禁止本地中断

确保一组内核语句被当做一个临界区处理的主要机制之一：中断禁止。宏：local\_irq\_disable和local\_irq\_enable。

11、禁止和寄回哦可延迟函数

可延迟函数可能在不可预知的时间执行，因此必须保护可延迟函数访问数据结构避免竞争条件。一种简单的方式就是禁止CPU上的中断，没有中断处理程序激活，软中断就不能异步开始。宏：local\_bh\_disable和local\_bh\_enable。

12、内核数据同步访问

同步原语的选取取决于访问数据结构的内核控制路径种类。

|  |  |  |
| --- | --- | --- |
| 内核控制路径 | 单处理器保护 | 多处理器进一步保护 |
| 异常 | 信号量 | 无 |
| 中断 | 本地中断禁止 | 自旋锁 |
| 可延迟函数 | 无 | 无或自选锁 |
| 异常与中断 | 本地中断禁止 | 自旋锁 |
| 异常与可延迟函数 | 本地软中断禁止 | 自选锁 |
| 中断与可延迟函数 | 本地中断禁止 | 自旋锁 |
| 异常、中断与可延迟函数 | 本地中断禁止 | 自旋锁 |

# 第六章 定时测量

1、jiffies变量

jiffies变量是一个计数器，用来记录自系统启动以来产生的节拍总数，每次时钟中断发生时（每个节拍）它便加1。

2、xtime变量

xtime变量存放当前时间和日志，它是一个timespec类型的数据结构，该结构字段：tv\_sec和tv\_nsec。

xtime变量通常是每一个节拍更新一次

3、计时体系结构初始化

（1）初始化节拍（HZ=100）

early\_init接口中godnet\_timer\_reload获取HZ的时间间隔。

（2）start\_kernel中初始化定时器

init\_IRQ()：

接口调用machine\_desc->init\_irq()（= godnet\_gic\_init\_irq）

init\_timers()：

初始化定时器软中断open\_softirq(TIMER\_SOFTIRQ, run\_timer\_softirq)

hrtimers\_init()：

高精度定时初始化open\_softirq(HRTIMER\_SOFTIRQ, run\_hrtimer\_softirq)，需要内核配置CONFIG\_HIGH\_RES\_TIMERS后才支持。

timekeeping\_init()：

初始化xtime以及选择默认时钟源&clocksource\_jiffies

time\_init()：

可以理解为硬件定时器初始化machine\_desc->timer-> init()（init= godnet\_timer\_init），在init接口中设置定时器0/1的中断函数godnet\_timer\_irq和godnet\_freetimer\_irq。在godnet\_timer\_irq中调用timer0\_clockevent事件结构体执行timer\_set\_mode接口重新加载HZ的时间间隔。

4、更新时间和日期

用户程序从xtime变量中获取当前时间和日期。内核必须周期性的更新该变量，才能使它的值保持相当的精确。

调用流程如下：tick\_setup\_periodic—>tick\_set\_periodic\_handler—>tick\_handle\_periodic—>tick\_periodic—>do\_timer—>update\_wall\_time，更新xtime。

5、更新本地CPU统计数

单处理器系统上的全局时钟中断处理程序或多处理器上的本地时钟中断处理程序调用update\_process\_times函数来更新一些内核统计数。

调用流程tick\_periodic—>update\_process\_times，即上面时间更新完后更新此处。

执行如下几个内容：

（1）检查当前进程运行了多长时间account\_process\_tick

根据用户态还是内核态选择不同的函数account\_user\_time or account\_system\_time。

（2）激活定时器软中断

run\_local\_timers—>raise\_softirq(TIMER\_SOFTIRQ)

（3）回收RCU

rcu\_check\_callbacks

（4）scheduler\_tick

该函数使当前进程的时间片计数器减1，并检查计数器是否已减到0.

6、记录系统负载

do\_timer中调用calc\_global\_load来计算负载。

7、监管内核代码

Linu包含一个readpofiler的最低要求的代码监管器。主要作用：确定内核的”热点”——执行最频繁的内核代码片段。

tick\_periodic—>profile\_tick函数为代码监管器采集数据，每次tick产生时采集一次。采集的数据在/proc/profile中，通过readprofile获取。

Linux2.6开始包含另外一个功能强大些的oprofile工具，profile\_tick调用time\_notify函数来收集这个新监管器所使用的数据。

8、软定时器和延时函数

Linux考虑两种类型的定时器：动态定时器和间隔定时器，前者由内核使用，而间隔定时器由进程用户态创建。

（1）动态定时器存放在下列的数据结构

struct timer\_list {

/\*

\* All fields that change during normal runtime grouped to the

\* same cacheline

\*/

struct list\_head entry;

unsigned long expires; 给出定时器到期时间，时间用字节数表示。用于将定时器插入双向循环链表队列，链表根据该字段的值进行分组存放。

struct tvec\_base \*base;

void (\*function)(unsigned long); 定时器到期时执行函数的地址

unsigned long data; data字段指定传递给定时器函数参数，利用该字段可以使多个设备共用一个定时器处理函数，data存放各个设备ID即可。

int slack;

#ifdef CONFIG\_TIMER\_STATS

int start\_pid;

void \*start\_site;

char start\_comm[16];

#endif

#ifdef CONFIG\_LOCKDEP

struct lockdep\_map lockdep\_map;

#endif

};

（2）动态定时器数据结构

struct tvec\_base {

spinlock\_t lock;

struct timer\_list \*running\_timer;

unsigned long timer\_jiffies;

unsigned long next\_timer;

struct tvec\_root tv1;

struct tvec tv2;

struct tvec tv3;

struct tvec tv4;

struct tvec tv5;

} \_\_\_\_cacheline\_aligned;

这个数据结构是一个叫做tvec\_base的每CPU变量：包含NR\_CPUS个元素，系统中每个CPU各有一个。

（3）动态定时器处理

run\_timer\_softirq函数是与TIMER\_SOFTIRQ软中断请求相关的可延迟函数。

9、与定时测量相关的系统调用

（1）时间

time()：返回从1970年1月1日午夜开始走过的秒数

gettimeofday()：返回从1970年1月1日午夜开始走过的秒数及在前1秒内走过的微妙数。

time()系统调用被gettimeofday()取代。gettimeofday()系统条用由sys\_gettimeofday函数实现，调用do\_gettimeofday()。

（2）间隔定时器系统调用setitimer和alarm

两个内容：发送信号所必需的频率，或者如果只需要产生一个信号，则频率为空；在下一个信号被产生以前剩余的时间。

激活间隔定时器第一个参数指定采取的策略：

ITIMER\_REAL：真正过去的时间，进程接收SIGALRM信号。这个是利用动态定时器实现的。

ITIMER\_VIRTUAL：进程在用户态下花费的时间，进程接收SIGVTALRM信号

ITIMER\_PROF：进程即在用户态下又在内核态下所发花费的时间，进程接收SIGPROF信号。

# 第七章 进程调度

1、调度策略

定义：决定什么时候以怎样的方式选择一个新进程运行的这组规则。Linux的调度基于分时技术：多个进程以“时间多路复用”方式运行。

进程一般分为三类：交互式进程，批处理进程，实时进程。

2、调度算法

调度类型：

SCHED\_FIFO：先进先出实时进程。只要没有更高优先级的实时进程就一直运行直到主动退出

SCHED\_RR：时间片轮转实时进程。时间片用完了就调度

SCHED\_NORMAL：普通的分时进程。也是根据时间片运行（默认是该种方式，见INIT\_TASK宏定义初始化）

（1）普通进程（SCHED\_NORMAL）

每个普通进程都有自己静态优先级，内核使用范围：100~139，值越大静态优先级越低。新进程继承父进程的静态优先级，用户通过系统调用nice() or setpriority()来改变进程的静态优先级。

静态优先级决定了进程的基本时间片：（140 - 静态优先级）\* 20，若静态优先级<120；（140 – 静态优先级）\* 5，若静态优先级>=120。普通进程优先级的典型值见书本表格。

每个普通进程还有动态优先级，内核使用范围：100~139。动态优先级是调度程序在选择新进程来运行的时候使用的数。与静态优先级关系的经验公式：

动态优先级=max（100, min(静态优先级-bonus+5, 139)）

bonus的范围：0~10，该值根据进程的平均睡眠时间相关（平均睡眠时间是进程在睡眠状态所消耗的平均ns数）。具体关系见书本表格。

（2）实时进程

每个实时进程的实时优先级范围：1~99，值越大优先级越小。调度程序总是让优先级高的进程运行。通过系统调用sched\_setparam()和sched\_setcheduler()改变进程的实时优先级。当具有相同的最高优先级时，调度程序选择第一个出现在本地CPU的运行队列链表中的进程。

实时进程发生下列情况时，会被另外进程取代：

进程被一个更高实时优先级的实时进程抢占

进程执行阻塞操作并进入睡眠

进程停止或被杀死

进程通过系统调用sched\_yield()自愿放弃CPU

进程基于时间片轮转的实时进程，而且用完了时间片。

基于时间片轮转实时进程的基本时间片长度依赖进程的静态优先级，其关系见普通进程算法。

3、调度程序所使用的数据结构

进程链表链接所有的进程描述符，而运行队列链表链接所有可运行进程（TASK\_RUNNING状态）的进程描述符。

数据结构runqueue，linux2.6调度程序最重要的数据结构。系统每个CPU都有它自己的运行队列，所有runqueue结构存放在runqueues每CPU变量中。宏this\_rq()产生本地CPU运行队列的地址，宏cpu\_rq(n)产生索引为n的CPU运行队列地址。

4、调度程序使用函数

scheduler\_tick()：维持当前最新的time\_slice计数器

try\_to\_wake\_up：唤醒睡眠进程

recalc\_task\_prio：更新进程的动态优先级

schedule()：选择要被执行的新进程

load\_balance()：维持多处理器系统中运行队列的平衡

# 第八章 内存管理

内核函数申请内存：

\_\_get\_free\_pages或alloc\_pages从分区页框分配器中获得页框

Kmem\_cache\_alloc或kmalloc使用slab分配器为专用或通用对象分配块

Vmalloc获得非连续的内存区

后面补充

# 第九章 进程地址空间

当用户态进程请求动态内存时，并没有获得请求的页框，而仅仅获得对一个新的线程地区间使用权，而这一线性区间就成为了进程地址空间的一部分，这区间叫做线性区。基本流程：进程请求页—> 申请到线性区—>缺页程序推迟分配页框—>创建和删除整个地址空间。

确定一个进程当前所拥有的线性区是内核的基本任务，可以让缺页异常处理程序有效的区分引发这个异常处理程序的两种不同类型的无效线性地址：

（1）由编程错误引发的无效线性地址

（2）由缺页引发的无效线性地址，即使这个线性地址属于进程的地址空间，但是对应于这个地址的页框仍然有待分配。

进程描述符 —> 内存描述符 —> 线性区 （包含vm\_operation\_struct）

1、内存描述符

与进程地址空间相关的全部信息都包含在一个叫内存描述符的数据结构中mm\_struct。进程描述符task\_struct的mm字段指向这个结构。

进程0（idle进程）拥有第一个进程描述符，内存描述符，线程描述符等（这些第一个数据皆静态数据，即使用结构体定义变量永久存在），后续进程的上述信息动态申请并加入到进程0，形成链表。

（1）第一个进程描述符（src/arch/arm/kernel/init\_task.c）

struct task\_struct init\_task = INIT\_TASK(init\_task);

/\*

\* INIT\_TASK is used to set up the first task table, touch at

\* your own risk!. Base=0, limit=0x1fffff (=2MB)

\*/

#define INIT\_TASK(tsk) \

{ \

.state = 0, \

.stack = &init\_thread\_info, \

.usage = ATOMIC\_INIT(2), \

.flags = PF\_KTHREAD, \

.prio = MAX\_PRIO-20, \

.static\_prio = MAX\_PRIO-20, \

.normal\_prio = MAX\_PRIO-20, \

.policy = SCHED\_NORMAL, \

.cpus\_allowed = CPU\_MASK\_ALL, \

.mm = NULL, \

.active\_mm = &init\_mm, \

.se = { \

.group\_node = LIST\_HEAD\_INIT(tsk.se.group\_node), \

}, \

.rt = { \

.run\_list = LIST\_HEAD\_INIT(tsk.rt.run\_list), \

.time\_slice = HZ, \

.nr\_cpus\_allowed = NR\_CPUS, \

}, \

.tasks = LIST\_HEAD\_INIT(tsk.tasks), \

INIT\_PUSHABLE\_TASKS(tsk) \

.ptraced = LIST\_HEAD\_INIT(tsk.ptraced), \

.ptrace\_entry = LIST\_HEAD\_INIT(tsk.ptrace\_entry), \

.real\_parent = &tsk, \

.parent = &tsk, \

.children = LIST\_HEAD\_INIT(tsk.children), \

.sibling = LIST\_HEAD\_INIT(tsk.sibling), \

.group\_leader = &tsk, \

RCU\_INIT\_POINTER(.real\_cred, &init\_cred), \

RCU\_INIT\_POINTER(.cred, &init\_cred), \

.comm = "swapper", \

.thread = INIT\_THREAD, \

.fs = &init\_fs, \

.files = &init\_files, \

.signal = &init\_signals, \

.sighand = &init\_sighand, \

.nsproxy = &init\_nsproxy, \

.pending = { \

.list = LIST\_HEAD\_INIT(tsk.pending.list), \

.signal = {{0}}}, \

.blocked = {{0}}, \

.alloc\_lock = \_\_SPIN\_LOCK\_UNLOCKED(tsk.alloc\_lock), \

.journal\_info = NULL, \

.cpu\_timers = INIT\_CPU\_TIMERS(tsk.cpu\_timers), \

.fs\_excl = ATOMIC\_INIT(0), \

.pi\_lock = \_\_RAW\_SPIN\_LOCK\_UNLOCKED(tsk.pi\_lock), \

.timer\_slack\_ns = 50000, /\* 50 usec default slack \*/ \

.pids = { \

[PIDTYPE\_PID] = INIT\_PID\_LINK(PIDTYPE\_PID), \

[PIDTYPE\_PGID] = INIT\_PID\_LINK(PIDTYPE\_PGID), \

[PIDTYPE\_SID] = INIT\_PID\_LINK(PIDTYPE\_SID), \

}, \

.thread\_group = LIST\_HEAD\_INIT(tsk.thread\_group), \

.dirties = INIT\_PROP\_LOCAL\_SINGLE(dirties), \

INIT\_IDS \

INIT\_PERF\_EVENTS(tsk) \

INIT\_TRACE\_IRQFLAGS \

INIT\_LOCKDEP \

INIT\_FTRACE\_GRAPH \

INIT\_TRACE\_RECURSION \

INIT\_TASK\_RCU\_PREEMPT(tsk) \

}

（2）第一个线程描述符（src/arch/arm/kernel/init\_task.c）

#define init\_thread\_info (init\_thread\_union.thread\_info)

#define init\_stack (init\_thread\_union.stack)

union thread\_union init\_thread\_union \_\_init\_task\_data ={ INIT\_THREAD\_INFO(init\_task) };

#define INIT\_THREAD\_INFO(tsk) \

{ \

.task = &tsk, \

.exec\_domain = &default\_exec\_domain, \

.flags = 0, \

.preempt\_count = INIT\_PREEMPT\_COUNT, \

.addr\_limit = KERNEL\_DS, \

.cpu\_domain = domain\_val(DOMAIN\_USER, DOMAIN\_MANAGER) | \

domain\_val(DOMAIN\_KERNEL, DOMAIN\_MANAGER) | \

domain\_val(DOMAIN\_IO, DOMAIN\_CLIENT), \

.restart\_block = { \

.fn = do\_no\_restart\_syscall, \

}, \

}

（3）第一个内存描述符（src/mm/init-mm.c）

struct mm\_struct init\_mm = {

.mm\_rb = RB\_ROOT,

.pgd = swapper\_pg\_dir,

.mm\_users = ATOMIC\_INIT(2),

.mm\_count = ATOMIC\_INIT(1),

.mmap\_sem = \_\_RWSEM\_INITIALIZER(init\_mm.mmap\_sem),

.page\_table\_lock = \_\_SPIN\_LOCK\_UNLOCKED(init\_mm.page\_table\_lock),

.mmlist = LIST\_HEAD\_INIT(init\_mm.mmlist),

INIT\_MM\_CONTEXT(init\_mm)

};

swapper\_pg\_dir：主内核页全局目录由主内存描述符的pgd字段指向，而主内存描述符存放在init\_mm变量。

#define KERNEL\_RAM\_VADDR (PAGE\_OFFSET + TEXT\_OFFSET)

.equ swapper\_pg\_dir, KERNEL\_RAM\_VADDR - 0x4000，

/\*说明：

\* swapper\_pg\_dir is the virtual address of the initial page table.

\* We place the page tables 16K below KERNEL\_RAM\_VADDR. Therefore, we must

\* make sure that KERNEL\_RAM\_VADDR is correctly set. Currently, we expect

\* the least significant 16 bits to be 0x8000, but we could probably

\* relax this restriction to KERNEL\_RAM\_VADDR >= PAGE\_OFFSET + 0x4000.

\*/

struct vm\_area\_struct \* mmap;字段指向属于该进程内存描述符的线性区链表头。

struct rb\_root mm\_rb; 字段指向属于该进程内存描述符的线性区红黑树根

struct vm\_area\_struct \* mmap\_cache; 保存进程最后一次引用线性区的描述符地址，减少查找一个给定线性地址所在线性区而花费时间。

mm\_users字段存放共享mm\_struct数据结构的轻量级进程个数，mm\_count字段是内存描述符的主使用计数器，在mm\_users次使用计数器中的所有用户在mm\_count中只作为一个单位。

2、内核线程的内存描述符

内核线程不用线性区，它使用一组最近运行的普通进程的页表。因而，每个进程描述符包含两种内存描述符指针:mm和active\_mm。mm字段指向进程所拥有的内存描述符，active\_mm字段指向进程运行时使用的内存描述符。普通进程两个字段存放相同的指针，内核线程mm=NULL，active\_mm字段初始化为前一个运行的active\_mm值。

3、线性区描述符struct vm\_area\_struct

字段vm\_mm指向拥有这个区间的进程的mm\_struct内存描述符，进程所拥有的线性区从来不重叠。

unsigned long vm\_start; /\* Our start address within vm\_mm. \*/

unsigned long vm\_end; /\* The first byte after our end addresswithin vm\_mm. \*/

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

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

上面3个字段组成线性区双向链表

struct rb\_node vm\_rb;字段用于红黑树的数据

vm\_ops字段存放线性区的操作方法。

进程所拥有的所有线性区通过一个简单的链表链接在一起的，出现在链表中的线性区按内存地址的升序排列。

为了存放进程的线性区，linux即使用了链表，有使用了红黑树。这两种数据结构包含指向同一线性区描述符的指针。当插入或删除一个线性区描述符时，内核通过红-黑树搜索前后元素并用搜索结果快速更新链表而不扫描链表。一般来说，红黑树确定指定地址的线性区，链表通常扫描整个线性区集合时使用。

由vm\_flags决定线性区的页访问权限。

4、线性区处理

find\_vma()：查找给地地址的最邻近区

find\_vma\_intersection()：查找一个与给定地址区间相重叠的线性区

get\_unmapped\_area()：查找一个空闲地址区域，具体说明如下：

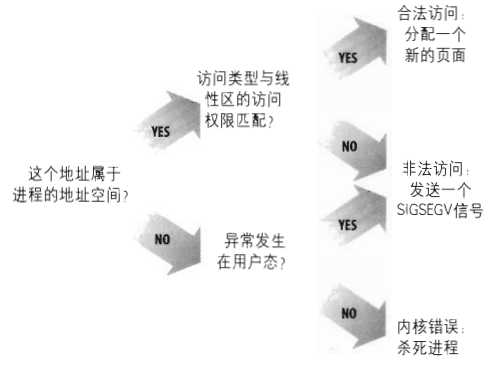
根据线性地址区间用于文件内存映射还是匿名内存映射，调用文件操作的file->f\_op->get\_unmapped\_area或内存描述符的current->mm->get\_unmapped\_area。后一种方法由函数arch\_get\_unmapped\_area()实现。每个进程有两种不同形式的线性区：一种从线性地址TASK\_SIZE/3开始并向高端地址增长；另外一种正好从用户堆栈开始并向低端地址增长。对于第一种用户态线性地址三分之一空间为有预定义起始线性地址的线性区而保留的（如：可执行文件的正文段、数据段和bss段）。

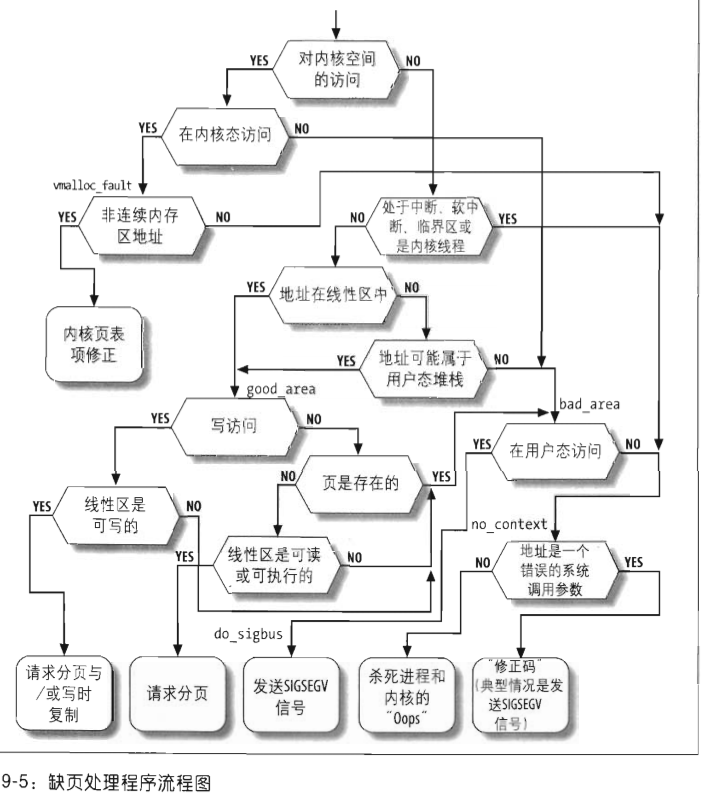
Insert\_vm\_struct()：向内存描述符链表插入一个线性区

5、分配线性地址区间do\_mmap和释放线性地址区间do\_munmap

6、缺页异常处理程序do\_page\_fault

基本流程图和详细处理流程图





7、创建和删除进程的地址空间

（1）创建

当创建一个新的进程时内核调用copy\_mm函数，通过建立新进程的所有页表和内存描述符来创建进程的地址空间。

每个进程都有自己的地址空间，但是轻量级进程可以通过clone()函数来创建，这些轻量级进程地址共享同一地址空间。使用clone时flag参数的CLONE\_VM标志被设置，则copy\_mm函数把父进程(current)地址空间给子进程(tsk)。

（2）删除

当进程结束时，内核调用exit\_mm()函数释放进程的地址空间。

8、堆的管理

每个进程都拥有一个特殊的线性区（所谓的堆heap），堆用于满足进程的动态内存请求。内存描述符的start\_brk与brk字段分别限定了区的开始地址和结束地址。使用brk()和mmap()系统调用实现。

# 第十章 系统调用

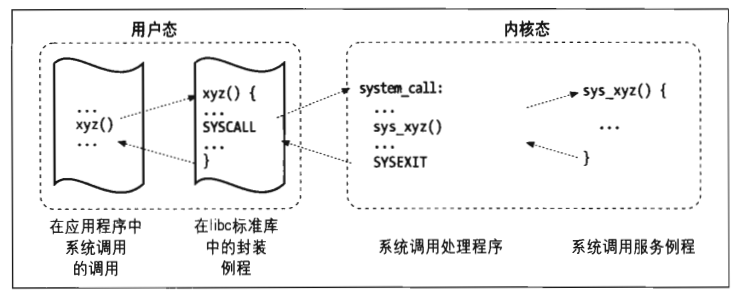
1、API和系统调用的区别

API只是一个函数定义，说明了如何获得一个给定的服务。系统调用通过软中断向内核态发出一个明确的请求。

POSIX标准针对API而不针对系统调用，判断一个系统是否与POSIX兼容要看它是否提供了一组合适的应用程序接口。

2、系统调用处理程序及服务例程

当用户的进程调用一个系统调用时，CPU切换到内核态并开始执行一个内核函数。



进程通过传递一个系统调用号的参数来识别所需的系统调用。为了系统调用号和相应的服务例程关联起来，内核利用一个系统调用分派表，这个表存放在sys\_call\_table数组中，有NR\_syscalls个表项。分派表中任意一个表项可以包含sys\_ni\_syscall函数地址表示“未实现”改表项对应的系统调用服务例程。

3、内核封装例程

系统调用主要由用户态进程使用，单也可以被内核线程调用，内核线程不能使用库函数，为了简化相应封装例程的声明，linux定义了7个从\_syscall0到\_syscall6的一组宏。每个宏名的数字0~6对应着封装例程所用的参数个数（系统调用号除外）。

# 第十一章 信号

1、信号作用

信号是很短的消息，可以被送到一个进程或一组进程，发给进程的唯一信息通常是一个数，名字前缀为SIG的一组宏来标识信号。信号两个主要目的：让进程知道已经发生一个特定的事件；强迫进程执行它自己代码中的信号处理程序。

常规信号：1~31，实时信号：32~64。同类型常规信号并不排队，一个常规信号被连续发送多次，只有其中一个发送到接收进程；实时信号必须排队以便发送的多个信号被接收到。

信号一个重要特点它们可以随时被发送给状态经常不可预知的进程。发送给非运行进程的信号必须由内核保存，直到进程恢复运行。内核区分信号传递的两个不同阶段：

（1）信号产生：内核更新目标进程的数据结构以表示一个新信号已经被发送

（2）信号传递：内核强迫目标进程通过以下方式对信号做出反应：或改变目标进程的执行状态；或开始执行一个特定的信号处理程序，或两者都是。

已经产生但还没有传递的信号称为挂起信号。

信号表现比较直观，但内核实现有些复杂：

（1）记住每个进程阻塞哪些信号

（2）从内核态切换到用户态时，对任何一个进程都要检查是否有一个信号已到达。

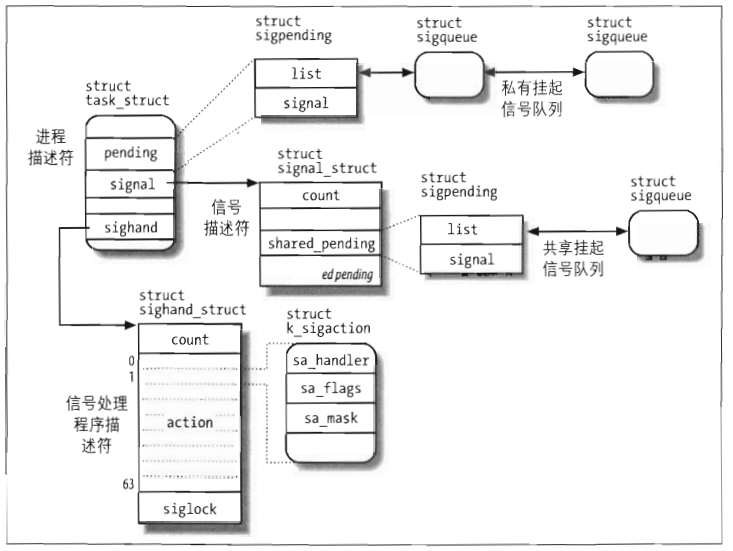
（3）确定是否可以忽略信号

（4）处理这样信号，信号可能在进程运行期间的任一时刻请求把进程切换到一个信号处理函数（用户态定义），并在这个函数返回以后恢复原来执行的上下文。

2、传递信号之前所执行的操作

SIGKILL和SIGSTOP信号不可以被显式的忽略、捕获或阻塞。

3、与信号相关的数据结构



（1）进程描述符中信号相关字段

struct task\_struct {

…..

struct signal\_struct \*signal; 指向进程信号描述符的指针

struct sighand\_struct \*sighand; 指向进程信号处理程序描述符的指针

sigset\_t blocked, real\_blocked; 被阻塞信号的掩码

sigset\_t saved\_sigmask; 被阻塞信号的临时掩码

struct sigpending pending; 存放私有挂起信号的数据结构

unsigned long sas\_ss\_sp; 信号处理程序备用堆栈地址

size\_t sas\_ss\_size; 备用堆栈大小

int (\*notifier)(void \*priv); 函数指针，设备驱动程序使用该函数阻塞进程的某些信号

void \*notifier\_data; 指向notifier函数可能使用的数据

sigset\_t \*notifier\_mask; 设备驱动程序通过notifier函数所阻塞的信号位掩码

….

}

typedef struct {

unsigned long sig[2];

} sigset\_t;

每个位代表一个信号类型，linux信号最大数为64。

（2）信号描述符

信号描述符被属于同一个线程组的所有进程共享，对属于同一线程组的每个进程而言，信号描述符中的字段必须是相同的。

struct signal\_struct {

atomic\_t sigcnt; 信号描述符的使用计数器

atomic\_t live; 线程组中活动进程的数量

int nr\_threads; 线程数

wait\_queue\_head\_t wait\_chldexit; 在系统调用wait4()中睡眠的进程等待队列

struct task\_struct \*curr\_target; 接收信号的线程组中最后一个进程的描述符

struct sigpending shared\_pending; 存放共享挂起的数据结构

int group\_exit\_code; 线程组的进程终止代码

int notify\_count; 杀死整个线程组的时候使用

struct task\_struct \*group\_exit\_task; 杀死整个线程组的时候使用

int group\_stop\_count; 停止整个线程组的时候使用

unsigned int flags; 传递修改进程状态的信号时使用的标志

…..

}

（3）信号处理程序描述符

每个信号必须怎样被线程组处理。

struct sighand\_struct {

atomic\_t count; 信号处理程序描述符的使用计数器

struct k\_sigaction action[64]; 在所传递信号上执行操作的结构数组

spinlock\_t siglock; 保护信号描述符合信号处理程序描述符自旋锁

wait\_queue\_head\_t signalfd\_wqh;

};

struct k\_sigaction {

struct sigaction sa;

};

struct sigaction {

\_\_sighandler\_t sa\_handler; 字段指定要执行操作的类型，指向信号处理程序的一个指针。

unsigned long sa\_flags; 标志集，指定必须怎样处理信号

\_\_sigrestore\_t sa\_restorer;

sigset\_t sa\_mask; 指定运行信号处理程序时要屏蔽的信号

};

（4）挂起信号队列

为了跟踪当前挂起信号是什么，内核把两个挂起信号队列与每个进程相关联

共享挂起信号队列：它位于信号描述符的shared\_pending字段，存放整个线程组的挂起信号

私有挂起信号队列：它位于进程描述符的pending字段，存放特定进程（轻量级进程）的挂起信号

4、在信号数据结构上的操作

内核使用几个函数和宏来处理信号

sigemptyset(set)和sigfillset(set)：把sigset\_t类型的变量中的位分别置0或置1。

sigaddset(set, nsig)和sigdelset(set, nisg)：把nsig信号在sigset\_t类型变量中的对应为分别置1或置0。

signal\_pending(p)：如果\*p进程描述符所表示的进程有非阻塞挂起信号，就返回值1，否则返回0。

rm\_from\_queue(mask,q)：从挂起信号队列q中删除与mask位掩码想对应的挂起信号。

flush\_sigqueue(q)：从挂起信号队列q中删除所有的挂起信号

flush\_signals(t)：删除发送给\*t进程描述符所表示的进程的所有信号。

….

5、产生信号

很多内核函数都会产生信号：完成信号第一步工作即根据需要更新一个或多个进程的描述符。不直接执行第二步信号传递操作，而是可能根据信号的类型和目标进程状态唤醒一些进程并促使这些进程接收信号。

（1）为进程产生信号的内核函数

send\_sig()、send\_sig\_info()：向单一进程发送信号

force\_sig()、force\_sig\_info()、force\_sig\_specific()：发送即不能被进程显示忽略，也不能被进程阻塞的信号。

sys\_tkill()：tkill的系统调用处理函数

sys\_tgkill()：tgkill()的系统调用处理函数

上面所有函数在结束时都调用specific\_send\_sig\_info()函数

（2）为线程组产生信号的内核函数

Send\_group\_sig\_info()：向某一个线程组发送信号，该线程组由它的一个成员进程的描述符来标识

kill\_pg()和kill\_pg\_info()：向一个进程组中的所有线程组发送信号

kill\_proc()和kill\_proc\_info()：向某一个线程组发送信号，该线程组由它的一个成员进程的PID来标识

sys\_kill()：kill()的系统调用处理函数

sys\_rt\_sigqueueinfo()：rt\_sigqueueinfo的系统调用处理函数

上面所有函数都会调用group\_send\_sig\_info()函数

（3）specific\_send\_sig\_info()函数

该函数想指定进程发送信号，三个参数：

Sig：信号编号，

info：或者是siginfo\_t表的地址，或者是三个特殊值中的一个：0：信号由用户态进程发送，1：意味着由内核发送，2：由内核发送的SIGSTOP或SIGKILL信号。

t：指向目标进程描述符的指针。

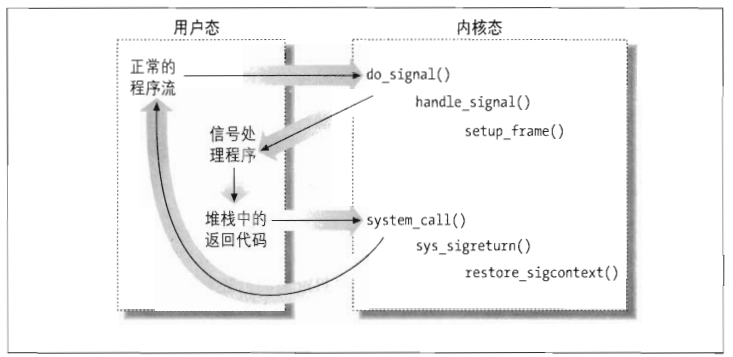
6、传递信号

为了处理非阻塞的挂起信号，内核调用do\_signal()函数，它接收两个参数：

regs：栈区地址，当前进程在用户态下寄存器内容存放在这个栈中

oldest：变量的地址，假设函数把阻塞信号的位掩码数组存放在这个变量中。

该函数通常只是在CPU要返回到用户态时才调用do\_signal()函数。



# 第十二章 虚拟文件系统