# 一、进程管理

## 1、进程描述符

### （1）进程与线程

进程是处于执行期的程序以及相关资源的总称。线程在linux上称为轻量级进程，没有独立的地址空间，一个进程下的所有线程共享地址空间、文件系统资源、文件描述符、信号处理程序等。

### （2）进程描述符task\_struct

内核把进程的列表存放在叫做任务队列的双向循环链表中。链表中的每一个项都是类型为task\_struct（即进程描述符的结构），它包含了一个具体进程的所有相关信息。通过slab分配器分配task\_struct结构，这样方便对象复用和缓存着色。该结构体有些大，下图只是指出其中比较重要的成员部分。



A．state状态

TASK\_RUNNING表示进程要么正在执行，要么正要准备执行。

    TASK\_INTERRUPTIBLE表示进程被阻塞（睡眠），直到某个条件变为真。条件一旦达成，进程的状态就被设置为TASK\_RUNNING。

    TASK\_UNINTERRUPTIBLE的意义与TASK\_INTERRUPTIBLE类似，除了不能通过接受一个信号来唤醒以外。

    \_\_TASK\_STOPPED表示进程被停止执行。

    \_\_TASK\_TRACED表示进程被debugger等进程监视。

    EXIT\_ZOMBIE表示进程的执行被终止，但是其父进程还没有使用wait()等系统调用来获知它的终止信息。

    EXIT\_DEAD表示进程的最终状态。

    EXIT\_ZOMBIE和EXIT\_DEAD也可以存放在exit\_state成员中

B．进程标识符

内核通过一个唯一的进程标示值或PID来标识每一个进程。

C．内核堆栈stack

内核通过thread\_union联合体来表示进程的内核栈，其中THREAD\_SIZE宏的大小为8192（首地址按照8192对齐），包含thread\_info大小，实际可用栈小于8192字节（见下节分析）。

当进程从用户态切换到内核态时，进程的内核栈总是空的，所以ARM的sp寄存器指向这个栈的顶端。因此，内核能够轻易地通过sp寄存器（因为对齐，低13位置0即首地址）获得当前正在CPU上运行的进程。

D．进程亲属关系成员

在Linux系统中，所有进程之间都有着直接或间接地联系，每个进程都有其父进程，也可能有零个或多个子进程。拥有同一父进程的所有进程具有兄弟关系。

    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。

    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。

    children表示链表的头部，链表中的所有元素都是它的子进程。

    sibling用于把当前进程插入到兄弟链表中。

    group\_leader指向其所在进程组的领头进程

E．进程调度

实时优先级范围是0到MAX\_RT\_PRIO-1（即99），而普通进程的静态优先级范围是从MAX\_RT\_PRIO到MAX\_PRIO-1（即100到139）。值越大静态优先级越低。

  static\_prio用于保存静态优先级，可以通过nice系统调用来进行修改。

  rt\_priority用于保存实时优先级。

  normal\_prio的值取决于静态优先级和调度策略。

  prio用于保存动态优先级。

  结构体成员policy表示进程的调度策略，目前主要有以下五种：

SCHED\_NORMAL用于普通进程，通过CFS调度器实现。

SCHED\_BATCH用于非交互的处理器消耗型进程

SCHED\_IDLE是在系统负载很低时使用。

SCHED\_FIFO（先入先出调度[算法](http://lib.csdn.net/base/31)）和SCHED\_RR（轮流调度算法）都是实时调度策略

F．进程地址空间

mm指向进程所拥有的内存描述符，而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。

G．信号处理

signal指向进程的信号描述符。

  sighand指向进程的信号处理程序描述符。

  blocked表示被阻塞信号的掩码，real\_blocked表示临时掩码。

  pending存放私有挂起信号的[数据结构](http://lib.csdn.net/base/31)。

  sas\_ss\_sp是信号处理程序备用堆栈的地址，sas\_ss\_size表示堆栈的大小。

  设备驱动程序常用notifier指向的函数来阻塞进程的某些信号（notifier\_mask是这些信号的位掩码），notifier\_data指的是notifier所指向的函数可能使用的数据

### （3）thread\_info

在2.6以前的内核中，各个进程的task\_struct存放在它们内核栈的尾端，目的是为了让那些像X86那样寄存器较少的硬件体系结构只要通过栈指针就能计算出它的位置，而避免使用额外的寄存器专门记录。现在用slab分配器动态生成task\_struct，所以只需在栈底（对于向下增长的栈来说）或栈顶（向上增长的栈来说）创建一个thread\_info，通过它指向该进程的task\_struct。内核栈的实现在上面已经提到，如下：

union thread\_union {

struct thread\_info thread\_info;

unsigned long stack[THREAD\_SIZE/sizeof(long)];

};

总大小8192，其中包含了thread\_info大小，实际可用栈<8192字节。

/\*

\* low level task data that entry.S needs immediate access to.

\* \_\_switch\_to() assumes cpu\_context follows immediately after cpu\_domain.

\*/

struct thread\_info {

unsigned long flags; /\* low level flags \*/

int preempt\_count; /\* 0 => preemptable, <0 => bug \*/

mm\_segment\_t addr\_limit; /\* address limit \*/

struct task\_struct \*task; /\* main task structure \*/指向task\_struct

struct exec\_domain \*exec\_domain; /\* execution domain \*/

\_\_u32 cpu; /\* cpu \*/

\_\_u32 cpu\_domain; /\* cpu domain \*/

struct cpu\_context\_save cpu\_context; /\* cpu context \*/存放CPU寄存器内容，在任务切换时加载或保存

\_\_u32 syscall; /\* syscall number \*/

\_\_u8 used\_cp[16]; /\* thread used copro \*/

unsigned long tp\_value;

#ifdef CONFIG\_CRUNCH

struct crunch\_state crunchstate;

#endif

union fp\_state fpstate \_\_attribute\_\_((aligned(8)));

union vfp\_state vfpstate;

#ifdef CONFIG\_ARM\_THUMBEE

unsigned long thumbee\_state; /\* ThumbEE Handler Base register \*/

#endif

struct restart\_block restart\_block;

};

struct cpu\_context\_save { //各个寄存器的值

\_\_u32 r4;

\_\_u32 r5;

\_\_u32 r6;

\_\_u32 r7;

\_\_u32 r8;

\_\_u32 r9;

\_\_u32 sl;

\_\_u32 fp;

\_\_u32 sp;

\_\_u32 pc;

\_\_u32 extra[2]; /\* Xscale 'acc' register, etc \*/

};

初始的task\_struct和thread\_info见下一章节分析，两者

（4）task\_struct和thread\_info关系



SP指针：就是从用户态切换到内核态时，SP寄存器指向的位置

cpu\_context：存放该进程的CPU寄存器内容，CPU切换到该进程时加载cpu\_context内容到CPU寄存器；如果该进程切换出去时，则将CPU寄存器内容保存到cpu\_context

## 2、进程创建

### （1）普通进程

Linux创建进程分解到两个单独的函数中去执行：fork()和exec()。首先，fork()通过拷贝当前进程创建一个子进程，子进程与父进程的区别仅仅在于PID、PPID和某些资源和统计量。exec()函数负责读取可执行文件并载入地址空间开始运行。

fork()使用写时拷贝页实现，那么实际开销就是复制父进程的页表以及子进程创建唯一的进程描述符。

vfork()与fork()功能相同，唯一的不同是vfork不拷贝父进程的页表项（理想情况下，尽量不要用vfork防止exec失败情况）。

fork()、vfork()和\_\_clone()库函数都根据各自需要的参数标志调用clone()系统调用，然后由clone调用内核do\_fork()接口。

do\_fork（kernel/fork.c）完成了创建中的大部分工作，主体实现函数copy\_process()。copy\_process主要完成以下工作：

A．调用dup\_task\_struct()为新进程创建一个内核栈、thread\_info结构和task\_struct，这些值与当前进程值相同。此时，子进程和父进程的描述符完全相同。

B．检查并确保新创建这个子进程后，当前用户拥有的进程数目没有超出给它分配的资源限制。

C．子进程着手使自己与父进程区别开来。

D．子进程状态被设置为TASK\_UNINTERRUPTIBLE，防止它投入运行

E．调用copy\_flags()更新task\_struct的flags成员

F．调用alloc\_pid()为新进程分配一个有效的PID

G．根据参数标志，拷贝或共享打开的文件、文件系统信息、信号处理函数、进程地址空间和命名空间等。

H．扫尾工作并返回一个指向子进程的指针。

再回到do\_fork()函数，如果copy\_process()函数成功返回，新创建的子进程被唤醒并让其投入运行，内核有意选择子程序首先执行。一般子进程都会马上调用exec()函数，这样可以避免写时拷贝的额外开销。

### （2）内核线程

内核经常需要在后台执行一些操作（如刷磁盘，空闲页回收等），这种任务可以交给内核线程。内核线程有以下几点不同：

没有独立的地址空间，所有内核线程共享地址空间。

只运行在内核态，从不切换到用户空间

只能使用大于PAGE\_OFFSET的线性地址空间

但是和普通进程一样，可以被调度，可以被抢占。

内核线程可以通过kernel\_thread和kthread\_create创建，两者主要不同点：

A．kthread\_create创建的内核线程有干净的上下文环境，适合于驱动模块或用户空间的程序创建内核线程使用，不会把某些内核信息暴露给用户程序。

B．二者创建的父进程不同：kernel\_thread创建的进程可以是init或其他内核线程，其中kernel\_init（1号进程）和kthreadd（2号进程）就是该方法创建；kthread\_ctreate创建的进程的父进程被指定kthreadd。

C．kthread\_create是kthread\_create\_on\_node的宏定义，该函数主要是将要创建的线程放到队列里，然后唤醒kthreadd内核线程调用create\_kthread接口创建，在完成创建之前kthread\_create\_on\_node一直等待。

### （3）进程0、1、2

所有进程的祖先叫做进程0（idle进程），它是在linux的初始化阶段从无到有创建的一个内核线程，唯一使用静态分配数据结构的进程（所有其他进程的数据结构都是动态分配），所以这些静态分配的数据结构也顺其自然的变成链表的头。在init/init\_task.c定义：

struct task\_struct init\_task = INIT\_TASK(init\_task) //进程描述符

.stack = &init\_thread\_info

.mm = NULL //内核线程没有独立内存地址空间，详细见进程地址空间一节

.active\_mm = &init\_mm

.tasks = LIST\_HEAD\_INIT(tsk.tasks) 自己指向自己

.real\_parent = &tsk

.parent = &tsk

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

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

.group\_leader = &tsk

.fs = &init\_fs

.files = &init\_files

.signal = &init\_signals

.sighand = &init\_sighand

union thread\_union init\_thread\_union \_\_init\_task\_data = { INIT\_THREAD\_INFO(init\_task) }; 在《linux内核之启动过程详解》一文中讲过在进入start\_kernel之前的汇编代码，将sp堆栈指向这个静态内核堆栈。

#define INIT\_THREAD\_INFO(tsk) \

{ \

.task = &tsk, //指向进程描述符 \

.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, \

}, \

}

主内核页全局目录存放在swapper\_pg\_dir中，该变量arch/arm/kernel/head.S中开头出定义，细节见《linux内核之启动过程详解》一文。

Start\_kernel完成内核相关的所有初始化之后，最后执行rest\_init函数，该函数会创建两个线程：

kernel\_thread(kernel\_init, NULL, CLONE\_FS | CLONE\_SIGHAND);

pid = kernel\_thread(kthreadd, NULL, CLONE\_FS | CLONE\_FILES);

同进程0共享所有的内核数据结构。

kernel\_init线程调用run\_init\_process—> do\_execve系统调用装入可执行程序/sbin/init（ps命令看到的1号进程就是这个init，search\_binary\_handler完成init可执行文件加载，空间映射等），完成内核态到用户态的切换，拥有自己的地址空间和数据结构，执行/etc下面的各种脚本，实现程序启动。

Kthreadd线程不停查看内核线程创建队列，看是否有内核线程需要创建，这也就有了上面kthread\_create内核线程的父进程都指向它。

在多处理器系统中，每个CPU都有一个进程0。开机时先启动CPU0，同时禁用其他CPU，在CPU0上初始化好后，再激活其他的CPU，这个激活过程在kernel\_init中完成。

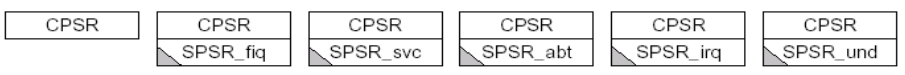
## 3、进程切换

### （1）arm简介

ARM微处理器共有37个32位寄存器，其中31个为通用寄存器，6个为状态寄存器。但是这些寄存器不能被同时访问，具体哪些寄存器是可以访问的，取决ARM处理器的工作状态及具体的运行模式。但在任何时候，通用寄存器R14~R0、程序计数器PC、一个状态寄存器都是可访问的。

通用寄存器：不分组寄存器（R0-R7） 、分组寄存器（R8-R14）（r13-sp，r14-lr） 、程序计数器R15（PC）。如下图所示：





Arm在linux下一般运行在两个模式：usr（用户态）和svc（内核态）。从上图可知usr和svc有各自独立的r13(sp)、r14(lr)、SPSR寄存器，这样用户空间和内核空间可以保存各自的堆栈，互不影响，从而方便的使进程从用户空间切换到内核空间。

不同进程的切换只在内核态进行，可以通过系统调用、异常、中断方式进入内核态。

### （2）不同进程间切换

进程切换：为了控制进程的执行，内核必须有能力挂起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行。尽管每个进程可以拥有属于自己的地址空间，但所有进程必须共享CPU寄存器。因此，在恢复一个进程的执行前，必须确保每个寄存器装入挂起进程时的值，这组寄存器数据称为硬件上下文。

从本质上说，每个进程切换由两部组成：

A．切换页全局目录以安装一个新的地址空间（在进程地址空间一节会有讲述）

B．切换内核态堆栈和硬件上下文，因为硬件上下文提供了内核执行新进程所需的所有信息，包含CPU寄存器。

针对A点，核心函数schedule() —> \_\_schedule() —> context\_switch() —> switch\_mm()

进程用户空间地址切换，更确切的说切换地址转换页表(pgd)，将下一个进程的pgd地址装入到CPU专门寄存器里，每个进程都有自身的页目录表pgd。由于pgd包括进程内核空间(0xc000 0000 ~ 0xffff ffff)和用户空间(0x0000 0000 ~ 0xbfff ffff)的地址映射，但是由于所有进程的内核空间的地址映射都是相同的。所以实质上就是进行用户空间的切换。如果下一个是内核线程切换则使用上一个线程的内核地址空间。

针对B点，核心函数schedule() —> \_\_schedule() —> context\_switch() —> switch\_to()（由各自平台实现）—> \_\_switch\_to()（在arm上这是一个汇编代码）源码如下。

\_\_switch\_to(prev, task\_thread\_info(prev), task\_thread\_info(next))

r0=prev //任务

r1= task\_thread\_info(prev) //内核栈

r2= task\_thread\_info(next) //内核栈

#define task\_thread\_info(task) ((struct thread\_info \*)(task)->stack)

/\*

\* Register switch for ARMv3 and ARMv4 processors

\* r0 = previous task\_struct, r1 = previous thread\_info, r2 = next thread\_info

\* previous and next are guaranteed not to be the same.

\*/

ENTRY(\_\_switch\_to)

UNWIND(.fnstart )

UNWIND(.cantunwind )

add ip, r1, #TI\_CPU\_SAVE //IP指向上一个线程thread\_info的cpu\_context成员地址

ldr r3, [r2, #TI\_TP\_VALUE] //r3=下一个线程tp\_value的值（即TLS寄存器）

//保存现场，存储r4 - sl, fp, sp, lr到上一个线程的cpu\_context里。

ARM( stmia ip!, {r4 - sl, fp, sp, lr} ) @ Store most regs on stack

THUMB( stmia ip!, {r4 - sl, fp} ) @ Store most regs on stack

THUMB( str sp, [ip], #4 )

THUMB( str lr, [ip], #4 )

#ifdef CONFIG\_CPU\_USE\_DOMAINS //未定义

ldr r6, [r2, #TI\_CPU\_DOMAIN]

#endif

set\_tls r3, r4, r5 //设置TLS寄存器，TLS线程局部存储

#if defined(CONFIG\_CC\_STACKPROTECTOR) && !defined(CONFIG\_SMP) //未定义

ldr r7, [r2, #TI\_TASK]

ldr r8, =\_\_stack\_chk\_guard

ldr r7, [r7, #TSK\_STACK\_CANARY]

#endif

#ifdef CONFIG\_CPU\_USE\_DOMAINS //未定义

mcr p15, 0, r6, c3, c0, 0 @ Set domain register

#endif

mov r5, r0 //临时保存上一个线程的task\_struct

add r4, r2, #TI\_CPU\_SAVE //r4指向下一个线程的cpu\_context

// thread\_notify\_head通知链

ldr r0, =thread\_notify\_head

mov r1, #THREAD\_NOTIFY\_SWITCH

bl atomic\_notifier\_call\_chain

#if defined(CONFIG\_CC\_STACKPROTECTOR) && !defined(CONFIG\_SMP) //未定义

str r7, [r8]

#endif

THUMB( mov ip, r4 ) //ip=r4：指向下一个线程的cpu\_context

mov r0, r5 //恢复r0重新指向上一个线程的task\_struct（未使用）

//恢复现场，将下一个线程寄存器加载到CPU，pc=cpu\_context->pc，刚好对应上面保存现场时的lr（即下一个线程要执行的地方）

ARM( ldmia r4, {r4 - sl, fp, sp, pc} ) @ Load all regs saved previously

THUMB( ldmia ip!, {r4 - sl, fp} ) @ Load all regs saved previously

THUMB( ldr sp, [ip], #4 )

THUMB( ldr pc, [ip] )

UNWIND(.fnend )

ENDPROC(\_\_switch\_to)

### （3）同一进程用户态和内核态切换

这里主要介绍系统调用引发的用户态和内核态切换，其他方式类似。

每个进程会有两个栈，一个用户栈，存在于用户空间，一个内核栈，存在于内核空间。当进程在用户空间运行时，CPU堆栈指针寄存器里面的内容是用户堆栈地址；当进程在内核空间时，CPU堆栈指针寄存器里的内容是内核栈空间地址，使用内核栈。

当进程通过系统调用陷入内核态时，进程使用的堆栈也要从用户栈转到内核栈。进程陷入内核态后，先把用户态堆栈的地址保存在内核栈中，然后设置堆栈指针寄存器的内容为内核栈的地址，这样就完成了用户栈向内核栈的转换；当进程从内核态恢复到用户态时，在内核态执行完后将保存在内核栈里面的用户栈的地址恢复到堆栈指针寄存器即可。这样就实现了内核栈和用户栈的互转。

在进程从用户态转到内核态的时候，进程的内核栈总是空的。这是因为，当进程在用户态运行时，使用的是用户栈，当进程陷入到内核态时，内核栈保存进程在内核态运行的相关信息，但是一旦进程返回到用户态后，内核栈中保存的信息无效，会全部恢复，因此每次进程从用户态陷入内核的时候得到的内核栈都是空的。所以在进程陷入内核的时候，直接把内核栈的栈顶地址给堆栈指针寄存器就可以。

进程调用系统调用通过swi指令产生中断发起内核服务请求，从而陷入内核。内核的相应入口点为ENTRY(vector\_swi)，执行这个函数前，硬件已经完成了如下事情：

A．将CPSR寄存器保存到SPSR\_svc寄存器中，将返回地址（用户空间执行swi指令的下一条指令）保存在lr\_svc。

B．设定CPSR寄存器的值。具体包括：CPSR.M = '10011'（svc mode），CPSR.I = '1'（disable IRQ），CPSR.IT = '00000000'（TODO），CPSR.J = '0'（）等

C．PC设定为swi异常向量的地址

源码（去掉无关宏）如下:

.align 5

ENTRY(vector\_swi)

sub sp, sp, #S\_FRAME\_SIZE //前面已经提到过用户态进入内核态，内核堆栈是空的，sp指向内核堆栈栈顶(这里sp\_svc，不是sp\_usr所以不会冲突，thread\_info在远离栈顶的低地址)

stmia sp, {r0 - r12} @ Calling r0 - r12 //r0-r12用户和内核共用，将用户空间的寄存器r0-r12压栈；而上一节内核线程切换CPU信息则保存thread\_info—>cpu\_context

ARM( add r8, sp, #S\_PC )

ARM( stmdb r8, {sp, lr}^ ) @ Calling sp, lr

THUMB( mov r8, sp )

THUMB( store\_user\_sp\_lr r8, r10, S\_SP ) @ calling sp, lr

mrs r8, spsr @ called from non-FIQ mode, so ok.

str lr, [sp, #S\_PC] @ Save calling PC //保存用户空间的lr即返回的pc地址到内核栈

str r8, [sp, #S\_PSR] @ Save CPSR //保存保存cpsr到内核堆栈

str r0, [sp, #S\_OLD\_R0] @ Save OLD\_R0

zero\_fp

#ifdef CONFIG\_ALIGNMENT\_TRAP

ldr ip, \_\_cr\_alignment

ldr ip, [ip]

mcr p15, 0, ip, c1, c0 @ update control register

#endif

enable\_irq

ct\_user\_exit

get\_thread\_info tsk //内核栈按照8192对齐，thread\_info在内核栈的首地址，所以通过sp的低13位设置为0编程首地址即thread\_info存放的地方。

//获取系统调用号

#if defined(CONFIG\_OABI\_COMPAT)

/\*

\* If we have CONFIG\_OABI\_COMPAT then we need to look at the swi

\* value to determine if it is an EABI or an old ABI call.

\*/

#ifdef CONFIG\_ARM\_THUMB

tst r8, #PSR\_T\_BIT

movne r10, #0 @ no thumb OABI emulation

USER( ldreq r10, [lr, #-4] ) @ get SWI instruction

#endif

#endif

adr tbl, sys\_call\_table @ load syscall table pointer 获取syscall表首地址

#if defined(CONFIG\_OABI\_COMPAT)

/\*

\* If the swi argument is zero, this is an EABI call and we do nothing.

\*

\* If this is an old ABI call, get the syscall number into scno and

\* get the old ABI syscall table address.

\*/

bics r10, r10, #0xff000000

eorne scno, r10, #\_\_NR\_OABI\_SYSCALL\_BASE

ldrne tbl, =sys\_oabi\_call\_table

#endif

local\_restart:

ldr r10, [tsk, #TI\_FLAGS] @ check for syscall tracing

stmdb sp!, {r4, r5} @ push fifth and sixth args 将swi后面的参数入栈

tst r10, #\_TIF\_SYSCALL\_WORK @ are we tracing syscalls?

bne \_\_sys\_trace

cmp scno, #NR\_syscalls @ check upper syscall limit

adr lr, BSYM(ret\_fast\_syscall) @ return address

ldrcc pc, [tbl, scno, lsl #2] @ call sys\_\* routine //执行相应的系统调用

add r1, sp, #S\_OFF

2: mov why, #0 @ no longer a real syscall

cmp scno, #(\_\_ARM\_NR\_BASE - \_\_NR\_SYSCALL\_BASE)

eor r0, scno, #\_\_NR\_SYSCALL\_BASE @ put OS number back

bcs arm\_syscall

b sys\_ni\_syscall @ not private func

#if defined(CONFIG\_OABI\_COMPAT) || !defined(CONFIG\_AEABI)

/\*

\* We failed to handle a fault trying to access the page

\* containing the swi instruction, but we're not really in a

\* position to return -EFAULT. Instead, return back to the

\* instruction and re-enter the user fault handling path trying

\* to page it in. This will likely result in sending SEGV to the

\* current task.

\*/

9001:

sub lr, lr, #4

str lr, [sp, #S\_PC]

b ret\_fast\_syscall //系统调用结束返回处理

#endif

ENDPROC(vector\_swi)

.align 5

/\*

\* This is the fast syscall return path. We do as little as

\* possible here, and this includes saving r0 back into the SVC

\* stack.

\*/

ret\_fast\_syscall:

UNWIND(.fnstart )

UNWIND(.cantunwind )

disable\_irq @ disable interrupts

ldr r1, [tsk, #TI\_FLAGS]

tst r1, #\_TIF\_WORK\_MASK

bne fast\_work\_pending

asm\_trace\_hardirqs\_on

/\* perform architecture specific actions before user return \*/

arch\_ret\_to\_user r1, lr //未实现

ct\_user\_enter

restore\_user\_regs fast = 1, offset = S\_OFF //恢复用户空间的CPU信息

UNWIND(.fnend )

//恢复用户空间CPU寄存器信息宏定义

.macro restore\_user\_regs, fast = 0, offset = 0

ldr r1, [sp, #\offset + S\_PSR] @ get calling cpsr //获取用户的CPSR

ldr lr, [sp, #\offset + S\_PC]! @ get pc //获取用户空间PC

msr spsr\_cxsf, r1 @ save in spsr\_svc

clrex @ clear the exclusive monitor

.if \fast

ldmdb sp, {r1 - lr}^ @ get calling r1 – lr //将之前压栈的用户寄存器出栈到寄存器

.else

ldmdb sp, {r0 - lr}^ @ get calling r0 - lr

.endif

mov r0, r0 @ ARMv5T and earlier require a nop

@ after ldm {}^

add sp, sp, #S\_FRAME\_SIZE - S\_PC //内核堆栈指针sp\_svc重新指向栈顶

movs pc, lr @ return & move spsr\_svc into cpsr //切换到usr返回到用户空间执行

.endm



进程切换只能发生在内核，切换时CPU的信息存放在thread\_info结构题的cpu\_context中；进程从用户态到内核态，进程的用户态CPU信息则存放在内核态堆栈空间pt\_regs地方。

## 4、进程销毁

进程终止的一般方式是调用exit()系统调用，即可能显式地调用，也可能隐式的从某个主函数返回；当进程接收到它既不能处理也不能忽略的信号或异常时，还可能被动终结。不管如何，在内核都是通过do\_exit完成工作。

调用了do\_exit()之后，尽管线程已经僵死不能再运行，但是系统还保留了它的进程描述符，这样可以让系统有办法在子进程终结后仍能获得它的信息。所以，进程终结时所需的清理工作和进程描述符的删除被分开执行。在父进程获得已终结的子进程的信息后，或者通知内核它并不关注那些信息后，子进程的task\_struct结构才被释放。

如果父进程在子进程之前退出，必须有机制来保证子进程能找到一个新的父进程，否则这些进程退出时永远处于僵死状态，白白耗费内存。对于这个问题，解决方法是给子进程在当前线程组内找一个线程作为父亲，如果不行，就让init做它们的父进程。

# 二、进程调度

进程调度程序可看作在可运行态进程之间分配有限的处理器时间资源的内核子系统。Linux提供了一种抢占式的多任务模式。

## 1、调度策略

Linux主要使用了以下几种调度策略（include/uapi/linux/sched.h）：

#define SCHED\_NORMAL 0

#define SCHED\_FIFO 1

#define SCHED\_RR 2

#define SCHED\_BATCH 3

/\* SCHED\_ISO: reserved but not implemented yet \*/

#define SCHED\_IDLE 5

SCHED\_NORMAL：普通调度策略（大部分进程属于这个），静态优先级100~139，对应的nice值从-20~19，代码里会进行转化，使用CFS调度类。

SCHED\_FIFO和SCHED\_RR：实时调度策略，静态优先级从0~99，使用rt调度类。

后两者用的比较少

rt调度类的优先级高于CFS调度类，因此使用实时调度策略的进程总是可以抢占普通调度策略进程。内核先从高优先级rt调度类中选择高优先级进程，再从低优先级CFS调度类中选择高优先级进程。

task\_struct中的policy表示上面的调度策略，fork时直接从父进程那里继承相应的策略，上面讲到进程0的task\_struct结构体是静态初始化，policy=0即NORMAL调度策略，通过系统调用sched\_setscheduler修改进程的调度策略。所以一般创建任务都是NORMAL调度策略。

top PR和NI：

两者都是表示进程的优先级，PR值是内核使用，NI是用户使用（nice –n xx<-20~19>）。对于上面的实时进程PR显示rt；对于普通进程PR则根据NI值进行转化（对应的静态优先级映射到100~139之间），其大致如下：

A．在Linux 2.6.23之前的版本中PR是一个动态值，在运行的过程中可能出现变化。大体策略是：如果一个进程sleep了比较多的时间，PR值会降低（即优先级提高）；如果一个进程占用了大量的CPU时间，PR值会升高（即优先级降低）。动态优先级算法：

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

bonus的值范围0~10，根据进程过往的平均睡眠时间确定，睡眠时间越长bonus越大。

B．在2.6.23版本之后，由于引进了CFS调度策略，不再简单根据一个进程sleep的时间动态调整其优先级了，PR值就固定为NI+20（刚好对应0~39用于获取下面数组prio\_to\_weight的权重，加上RT最小优先级100就映射到静态优先级100~139）。

## 2、调度算法

### （1）实时策略FIFO和RR

FIFO实现了一种简单、先入先出的调度算法：不使用时间片。这意味着一旦该类进程处于执行状态，就一直执行下去，直到它自己受阻塞或显示的释放处理器为止，只有更高优先级的SCHED\_FIFO或SCHED\_RR任务才能抢占它。处于可运行状态的该类进程会比任何SCHED\_NORMAL级的进程都先得到调度。

RR（实时轮转调度算法）可以说是带有时间片的SCHED\_FIFO。当SCHED\_RR进程耗尽它的时间片时，在同一级的其他实时进程被轮流调度。也就是说时间片耗完仍然处于可执行队列，只是保证同一优先级的其他进程可以执行，而低优先级想执行只能等到同一优先级所有进程都处于不可执行状态才有机会。

#define RR\_TIMESLICE (100 \* HZ / 1000) 所以默认是100ms

两种实时算法都是静态优先级0~99，内核不计算动态优先级，从而保证高优先级的进程总是能够抢占低优先级的进程。只有当高优先级不可执行后，低优先级才有机会执行。

### （2）普通策略CFS

CFS（完全公平调度）的基本原理：设定一个调度周期（sched\_latency\_ns），目标是让每个进程在这个周期内至少有机会运行一次，换一种说法就是每个进程等待CPU的时间最长不超过这个调度周期；然后根据进程的数量，大家平分这个调度周期内的CPU使用权；nice优先级不再用于计算时间片，而是作为进程获得的CPU运行比的权重，nice越大（优先级越低）权重越大，反之越小；每个进程的累计运行时间保存在自己的vruntime字段里，哪个进程的vruntime最小就获得本轮运行的权利。

调度周期：所有可运行状态进程都调度执行一遍的时间。涉及结构体数据：

struct task\_struct {

…

const struct sched\_class \*sched\_class; //指向调度器类链表

struct sched\_entity se; //cfs调度器实体结构

struct sched\_rt\_entity rt; //rt调度器实体结构

…

}

struct sched\_entity {

struct load\_weight load; /\* for load-balancing \*/

struct rb\_node run\_node; //红黑树

struct list\_head group\_node;

unsigned int on\_rq;

u64 exec\_start; //记录上一次更新的时间点

u64 sum\_exec\_runtime; //进程执行的总时间

u64 vruntime; //虚拟运行时间

u64 prev\_sum\_exec\_runtime; //进程切换进CPU是的sum\_exec\_runtime

u64 nr\_migrations;

…

}

注：相关运行时间都是ns为单位

now与exec\_start的delta用于计算sum\_exec\_runtime和vruntime

sum\_exec\_runtime与prev\_sum\_exec\_runtime的delta用于计算当前任务是否超过预期时间，超过则需要调度。

vruntime变量存放进程的虚拟运行时间，该运行时间的计算是经过了所有可运行进程总数的标准化，是CFS调度的核心数据。基本计算方法如下：

A．分配给进程的实际运行时间=调度周期 \* 进程权重 / 所有进程权重之和 （公式1）

sysctl\_sched\_latency：存放调度周期；每个nice都有一个权重值，内核变量定义如下：

static const int prio\_to\_weight[40] = {

/\* -20 \*/ 88761, 71755, 56483, 46273, 36291,

/\* -15 \*/ 29154, 23254, 18705, 14949, 11916,

/\* -10 \*/ 9548, 7620, 6100, 4904, 3906,

/\* -5 \*/ 3121, 2501, 1991, 1586, 1277,

/\* 0 \*/ 1024, 820, 655, 526, 423,

/\* 5 \*/ 335, 272, 215, 172, 137,

/\* 10 \*/ 110, 87, 70, 56, 45,

/\* 15 \*/ 36, 29, 23, 18, 15,

};

B．虚拟vruntime

vruntime = 实际运行时间 \* 1024 / 进程权重 （公式2）

1024是nice=0时的权重值，其他nice值的进程以该值为基准，也就是说nice=0的进程虚拟运行时间和实际运行时间相等，而nice越小优先级越高，虚拟运行时间越小被调度的机会就越多。

将公式1代入公式2：

vruntime = (调度周期 \* 进程权重 / 所有进程权重之和) \* 1024 / 进程权重

= 调度周期 \* 1024 / 所有进程权重之和 （公式3）

从该公式可以知道一个调度周期内，不同进程的vruntime相同，体现了该调度器完全公平一说。

C．虚拟运行时间和实际运行时间区别

从上面公式可以看到，在一个调度周期内，不同进程的虚拟运行时间是相同的，但真实时间是不同的。

从公式（2）可知：vruntime的增加量和进程权重成反比，权重越大（优先级越高）vruntime增长越少，反之越多。CFS总是选择vruntime最小的进程进行调度，所有权重越大得到cpu的机会越高。因此，理论上一个调度周期为了保持vruntime相同，进程权重越大的，那么实际运行时间也要越长。

说明：

新进程、休眠进程、进程切换CPU等vruntime都会进行相应的补偿，免得和当前相差太大，一直处于执行。

红黑树：

CFS调度算法的核心：选择具有最小的vruntime任务。CFS使用红黑树来组织可运行进程队列，并利用其迅速找到最小vruntime值的进程。

挑选下一个任务：schedule() -> pick\_next\_task() -> pick\_next\_task\_fair() -> pick\_next\_entity() -> \_\_pick\_next\_entity() -> rb\_entry()

向树中加入进程：enqueue\_entity()，在进程变为可运行装填或者fork第一次创建进程时

从树中删除进程：dequeue\_entity()，发生进程阻塞或者终止时。

proc信息：

目录：/proc/sys/kernel下面：

sched\_latency\_ns：sysctl\_sched\_latency，表示一个运行队列所有进程运行一次的周期（即上面的调度周期）如果进程数超过sched\_nr\_latency（这个变量不能通过/proc设置，它是由( sysctl\_sched\_latency + sysctl\_sched\_min\_granularity - 1) / sysctl\_sched\_min\_granularity确定的），那么调度周期就是sched\_min\_granularity\_ns\*运行队列里的进程数，与sysctl\_sched\_latency无关；否则队列进程数小于sched\_nr\_latency，运行周期就是sysctl\_sched\_latency。

sched\_min\_granularity\_ns：sysctl\_sched\_min\_granularity，表示进程最少运行时间，防止频繁的切换，对于交互系统（如桌面），该值可以设置得较小，这样可以保证交互得到更快的响应

说明：

如果需要调度的进程个数为n，那么平均每个进程占用的CPU时间为sched\_latency\_ns／n。显然，每个进程实际占用的CPU时间会因为n的增大而减小。但是实现上不可能让它无限的变小，所以sched\_min\_granularity\_ns的值也限定了每个进程可以获得的执行时间周期的最小值。当进程很多，导致使用了sched\_min\_granularity\_ns作为最小调度周期时，对应的调度延时也就不在遵循sched\_latency\_ns的限制，而是以实际的需要调度的进程个数n \* sched\_min\_granularity\_ns进行计算。当然，我们也可以把这理解为CFS的”时间片”，不过我们还是要强调，CFS是没有跟O1类似的“时间片“的概念的

sched\_wakeup\_granularity\_ns：sysctl\_sched\_wakeup\_granularity，该变量表示进程被唤醒后至少应该运行的时间的基数，它只是用来判断某个进程是否应该抢占当前进程，并不代表它能够执行的最小时间（sysctl\_sched\_min\_granularity），如果这个数值越小，那么发生抢占的概率也就越高

sched\_features：sysctl\_sched\_features，该变量表示调度器支持的特性

sched\_child\_runs\_first：sysctl\_sched\_child\_runs\_first，该变量表示在创建子进程的时候是否让子进程抢占父进程，即使父进程的vruntime小于子进程，这个会减少公平性，但是可以降低write\_on\_copy，具体要根据系统的应用情况来考量使用哪种方式。

sched\_rt\_period\_us / sched\_rt\_runtime\_us：sysctl\_sched\_rt\_period/ sysctl\_sched\_rt\_runtime，两个参数一起决定了实时进程在以sysctl\_sched\_rt\_period为周期的时间内，实时进程最多能够运行的总的时间不能超过sysctl\_sched\_rt\_runtime。

sched\_compat\_yield：sysctl\_sched\_compat\_yield，该参数可以让sched\_yield()系统调用更加有效，让它使用更少的cpu，对于那些依赖sched\_yield来获得更好性能的应用可以考虑设置它为1

sched\_migration\_cost：sysctl\_sched\_migration\_cost该变量用来判断一个进程是否还是hot，如果进程的运行时间（now - p->se.exec\_start）小于它，那么内核认为它的code还在cache里，所以该进程还是hot，那么在迁移的时候就不会考虑它

sched\_nr\_migrate：sysctl\_sched\_nr\_migrate，在多CPU情况下进行负载均衡时，一次最多移动多少个进程到另一个CPU上

sched\_tunable\_scaling：sysctl\_sched\_tunable\_scaling，当内核试图调整sysctl\_sched\_min\_granularity，sysctl\_sched\_latency和sysctl\_sched\_wakeup\_granularity这三个值的时候所使用的更新方法，0为不调整，1为按照cpu个数以2为底的对数值进行调整，2为按照cpu的个数进行线性比例的调整。

默认是1，CPU 4核为例：

sysctl\_sched\_latency = 6ms \* (1 + log(4)) = 18ms

sysctl\_sched\_min\_granularity =

## 3、调度实现

### （1）调度函数

进程调度的主要入口函数是schedule（kernel/sched/core.c），它选择哪个进程可以运行，何时将其投入运行。它会找到一个最高优先级的调度类，然后从该类中选择优先级最高的进程。调用到schedule函数的情况：

A．主动阻塞情况：mutex, semaphore, waitqueue等

B．进程运行时间用完情况：tick定时器中断调用scheduler\_tick()，如果当前进程本次执行时间用完则会设置TIF\_NEED\_RESCHED，退出中断后会进行调度

C．主动调用wake\_up唤醒进程，只是将唤醒的进程放入可执行队列，如果该进程比当前运行的进程优先级高，则会设置TIF\_NEED\_RESCHED，在下面情况下发生调度：

抢占式内核（见<linux内核之中断和异常.docx>的图表例子）：

在系统调用或异常上下文，在下一个遇到preempt\_enable时就会发生调度（大部分是快执行结束时发生）

在中断上下文，中断返回后进入抢占上下文，开始判断是否要调度

非抢占式内核：

主动调用cond\_resched()

主动调用schedule()

系统调用和异常处理结束，返回用户空间前查TIF\_NEED\_RESCHED是否要调度

中断处理结束，返回用户空间前检查TIF\_NEED\_RESCHED是否要调度

主要由接口pick\_next\_task实现：

static inline struct task\_struct \* pick\_next\_task(struct rq \*rq)

{

const struct sched\_class \*class;

struct task\_struct \*p;

/\*

\* Optimization: we know that if all tasks are in

\* the fair class we can call that function directly:

\*/

if (likely(rq->nr\_running == rq->cfs.h\_nr\_running)) {

p = fair\_sched\_class.pick\_next\_task(rq);

if (likely(p))

return p;

}

for\_each\_class(class) {

p = class->pick\_next\_task(rq);

if (p)

return p;

}

}

该函数会以优先级为序，从高到低，依次检查每一个调度类，并且从最高优先级的调度类中，选择最高优先级的进程。

上面先执行了CFS调度类查找，这样做的主要目的：系统运行的绝大多数进程都是普通进程，这样可以加速选择下一个CFS进程。

### （2）睡眠和唤醒

A．休眠

休眠的进程处于一个特殊的不可执行状态。进程休眠有多种原因，但肯定都是为了等待一些事件。对于内核操作都是相同：进程把自己标记成休眠状态，从可执行红黑树中移出，放入等待队列，然后调用schedule()选择和执行一个其他进程。休眠通过等待队列进行处理，等待队列是由等待某些事件发生的进程组成的简单链表。

B．唤醒

唤醒：进程被设置为可执行状态，然后在从等待队列中移到可执行的红黑树中。通过函数wake\_up系列进行，它会唤醒指定的等待队列上的某个或所有进程，如果被唤醒的进程优先级比当前正在执行的进程优先级高，还要设置need\_resched标志。

### （3）抢占

内核提供了一个need\_resched标志来表明是否需要重新执行一次调度。当某个进程应该被抢占时，scheduler\_tick()会设置；当一个优先级高的进程进入可执行状态，try\_to\_wake\_up()会设置；返回用户空间以及从中断返回的时候，内核也会检查need\_resched标志。

A．用户抢占

内核即将返回用户空间的时候，如果need\_resched标志被设置，会导致schedule()被调用，此时就会发生用户抢占。在内核返回用户空间的时候，知道自己是安全的，因此就可以选择一个新的进程去执行。

用户抢占发生在以下情况：

从系统调用返回用户空间

从中断处理程序返回用户空间

B．内核抢占

只要重新调度是安全的，内核就可以在任何时间抢占正在执行的任务。只要没有持有锁，重新调度就是安全的，内核就可以进行抢占。锁是非抢占区域的标志。

内核抢占发生在以下情况：

中断处理程序正在执行，且返回内核空间之前

内核代码再一次具有可抢占性的时候

如果内核中的任务显式调用schedule()

如果内核中的任务阻塞（这同样也会导致调用schedule）

### （4）多CPU的CFS调度

现在CPU一般有多个核，CFS从一开始就考虑了这种情况，它对每个CPU核心都维护一个调度队列，这样每个CPU都对自己的队列进程调度即可。这也是CFS比O1调度算法更高效的根本原因：每个CPU一个队列，就可以避免对全局队列使用大内核锁，从而提高了并行效率。当然，这样最直接的影响就是CPU之间的负载可能不均，为了维持CPU之间的负载均衡，CFS要定期对所有CPU进行load balance操作，于是就有可能发生进程在不同CPU的调度队列上切换的行为。这种操作的过程也需要对相关的CPU队列进行锁操作，从而降低了多个运行队列带来的并行性。不过总的来说，CFS的并行队列方式还是要比O1的全局队列方式要高效。尤其是在CPU核心越来越多的情况下，全局锁的效率下降显著增加。

CFS对多个CPU进行负载均衡的行为是idle\_balance()函数实现的，这个函数会在CPU空闲的时候由schedule()进行调用，让空闲的CPU从其他繁忙的CPU队列中取进程来执行。我们可以通过查看/proc/sched\_debug的信息来查看所有CPU的调度队列状态信息以及系统中所有进程的调度信息。内容较多，我就不在这里一一列出了，有兴趣的同学可以自己根据相关参考资料（最好的资料就是内核源码）了解其中显示的相关内容分别是什么意思。

在CFS对不同CPU的调度队列做均衡的时候，可能会将某个进程切换到另一个CPU上执行。此时，CFS会在将这个进程出队的时候将vruntime减去当前队列的min\_vruntime，其差值作为结果会在入队另一个队列的时候再加上所入队列的min\_vruntime，以此来保持队列切换后CPU队列的相对公平

# 三、进程地址空间

## 1、地址空间

在Documentation/arm/memory.txt文件中定义了arm平台的线性地址空间布局情况，其与物理内存关系图：



详细说明见：《linux内核之虚拟内存》

进程的地址空间由进程可寻址的虚拟内存组成。每个进程都有自己独立的地址空间，两个进程之间没有关系，互不影响。这些虚拟内存区域包含各种对象，常见的有如下：

代码段：可执行文件代码的内存映射

数据段：可执行文件的已初始化全局变量的内存映射

Bss段：包含未初始化的全局变量，也称bss段零页的内存映射

用户进程用户空间栈的零页内存映射，栈大小编译时指定

每个共享库的代码段、数据段和bss段也会被载入进程的地址空间

任何内存映射文件

任何共享内存段

任何匿名的内存映射，比如malloc

进程地址空间中任何有效地址都只能位于唯一的区域，这些内存区域不能相互覆盖。

## 2、内存描述符

### （1）数据结构

内核使用内存描述符结构体标识进程的地址空间，该数据结构包含了和进行地址空间相关的全部信息。

struct task\_struct {

…

struct mm\_struct \*mm, \*active\_mm; //进程任务描述符指向内存描述符，mm为进程地址空间所有信息，active\_mm则为当前执行的地址空间信息

…

}

struct mm\_struct {

struct vm\_area\_struct \* mmap; //内存区域链表

struct rb\_root mm\_rb; //内存区域形成的红黑树

struct vm\_area\_struct \* mmap\_cache; //最近使用的内存区域

#ifdef CONFIG\_MMU

unsigned long (\*get\_unmapped\_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);

#endif

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

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

unsigned long cached\_hole\_size; /\* if non-zero, the largest hole below free\_area\_cache \*/

unsigned long free\_area\_cache; /\* first hole of size cached\_hole\_size or larger \*/

unsigned long highest\_vm\_end; /\* highest vma end address \*/

pgd\_t \* pgd;

atomic\_t mm\_users; /\* How many users with user space? \*/

atomic\_t mm\_count; /\* How many references to "struct mm\_struct" (users count as 1) \*/

int map\_count; /\* number of VMAs \*/

spinlock\_t page\_table\_lock; /\* Protects page tables and some counters \*/

struct rw\_semaphore mmap\_sem;

struct list\_head mmlist; /\* List of maybe swapped mm's. These are globally strung

\* together off init\_mm.mmlist, and are protected

\* by mmlist\_lock

\*/

unsigned long hiwater\_rss; /\* High-watermark of RSS usage \*/

unsigned long hiwater\_vm; /\* High-water virtual memory usage \*/

unsigned long total\_vm; /\* Total pages mapped \*/

unsigned long locked\_vm; /\* Pages that have PG\_mlocked set \*/

unsigned long pinned\_vm; /\* Refcount permanently increased \*/

unsigned long shared\_vm; /\* Shared pages (files) \*/

unsigned long exec\_vm; /\* VM\_EXEC & ~VM\_WRITE \*/

unsigned long stack\_vm; /\* VM\_GROWSUP/DOWN \*/

unsigned long def\_flags;

unsigned long nr\_ptes; /\* Page table pages \*/

unsigned long start\_code, end\_code, start\_data, end\_data; //代码数据收尾地址

unsigned long start\_brk, brk, start\_stack; //堆的收尾地址，栈

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

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

/\*

\* Special counters, in some configurations protected by the

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

\*/

struct mm\_rss\_stat rss\_stat;

struct linux\_binfmt \*binfmt;

cpumask\_var\_t cpu\_vm\_mask\_var;

/\* Architecture-specific MM context \*/

mm\_context\_t context;

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

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

…

}

mmap和mm\_rb指向的是相同数据对象（vm\_area\_stuct虚拟内存区域），只是组织方式不一样。mmap以链表形式存放，利于简单、高效的遍历所有元素；mm\_rb以红黑树的形式存放，更适合搜索指定元素。

pgd为该进程的页全局目录，实现虚拟地址和物理内存的映射，在进程切换时由switch\_mm函数将pgd首地址放入CPU寄存器供mmu（硬件实现）使用。TLB则是pgd的一个子集，存放已经使用过的映射，加速地址映射查找。

mm\_users和mm\_count：mm\_users记录正在使用该内存的进程数目；mm\_count域是mm\_struct结构体的主引用计数。区别：mm\_users一个进程创建多个线程时会增加，mm\_count当内核线程共享进程地址空间时增加，初始化是1。

所有的mm\_struct结构体都通过自身的mmlist域连接在一个双向链表中，该链表的首元素是init\_mm内存描述符，它代表进程0的地址空间。

见后面关联图说明。

### （2）分配与撤销

A．分配

在进程创建一节提到过，do\_fork的copy\_process()完成大部分工作。copy\_process()中的copy\_mm()实现复制父进程的内存描述符，源码解析如下：

static int copy\_mm(unsigned long clone\_flags, struct task\_struct \*tsk)

{

struct mm\_struct \*mm, \*oldmm;

int retval;

tsk->min\_flt = tsk->maj\_flt = 0;

tsk->nvcsw = tsk->nivcsw = 0;

/\* 把子进程初始化成NULL \*/

tsk->mm = NULL;

tsk->active\_mm = NULL;

/\* 由于内核线程没有独立的地址空间mm=NULL，所以当前current(即父进程)的mm为NULL，则表示父进程是内核线程，子进程上面已经初始化为NULL，这里直接返回 \*/

oldmm = current->mm;

if (!oldmm)

return 0;

/\* 如果设置了CLONE\_VM共享地址空间标识，则子进程直接指向父进程的内存描述符，使用计数+1，一般用户创建线程使用 \*/

if (clone\_flags & CLONE\_VM) {

atomic\_inc(&oldmm->mm\_users);

mm = oldmm;

goto good\_mm;

}

retval = -ENOMEM;

/\* 其他情况则是通过allocate\_mm()申请自己的mm\_struct，拷贝父进程的内容，再进行自己独立的初始化（包括pgd页表申请等）

allocate\_mm()是kmem\_cache\_alloc的宏，从mm\_cashep slab缓存中分配得到

并申请页表，页表的用户部分未进行映射清0，而内核部分则直接拷贝init\_mm的swapp\_pgd\_dir页表（细节见《虚拟内存管理》文档）。

\*/

mm = dup\_mm(tsk);

if (!mm)

goto fail\_nomem;

good\_mm:

tsk->mm = mm;

tsk->active\_mm = mm;

return 0;

fail\_nomem:

return retval;

}

B．撤销

当进程退出时，内核会调用exit\_mm()函数，该函数执行一些常规的撤销工作，同时更新一些统计量。其中，该函数会调用mmput减少mm\_users计数，如果到0，调用mmdrop函数减少mm\_count计数，如果到0，那么调用free\_mm()宏通过kmem\_cache\_free()函数将mm\_struct结构体归还到mm\_cachep slab缓存中。

### （3）内核线程mm\_struct

上面分配一节已经看到，内核线程对应的进程描述符中mm域为空，这是因为内核线程在用户空间中没有任何页，所以不需要有自己的内存描述符和页表，所有内核线程共享内核地址空间。

尽管mm域为空，即使访问内核内存，内核线程也还是需要使用一些数据的。为了避免内核线程为内存描述符和页表浪费内存，避免向新地址空间进程切换浪费处理器周期，内核线程将直接使用前一个进程的内存描述符，进程切换源码如下：

static inline void context\_switch(struct rq \*rq, struct task\_struct \*prev,

struct task\_struct \*next)

{

struct mm\_struct \*mm, \*oldmm;

…

mm = next->mm; //下一个将执行进程的mm

oldmm = prev->active\_mm; //当前执行进程的active\_mm

/\* mm=NULL则为内核线程，内核线程的active\_mm指向当前执行进程的active\_mm，同时mm\_count+1

mm != NULL则为用户进程，通过switch\_mm实现内存描述符的切换，主要完成将pgd首地址放入CPU寄存器供mmu（硬件实现）使用

\*/

if (!mm) {

next->active\_mm = oldmm;

atomic\_inc(&oldmm->mm\_count);

enter\_lazy\_tlb(oldmm, next);

} else

switch\_mm(oldmm, mm, next);

/\* 如果当前的进程为内核线程，则将active\_mm=NULL，下次运行再重新赋值 \*/

if (!prev->mm) {

prev->active\_mm = NULL;

rq->prev\_mm = oldmm;

}

…

}

## 3、虚拟内存区域

### （1）数据结构

虚拟内存区域（VMA）由vm\_area\_struct结构体描述，指定线性地址空间内连续区间上的一个独立内存范围。每一个VMA作为一个单独的内存对象管理，具有一致的属性（比如访问权限等），因此，一个VMA就代表了一种类型的内存区域（如内存映射文件、进程用户空间栈等）。结构体如下：

struct vm\_area\_struct {

unsigned long vm\_start; //内存区间的收地址

unsigned long vm\_end; //内存区间的尾地址

//VMA双向链表，按照地址顺序排序

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

struct rb\_node vm\_rb; //放在红黑树上的节点

unsigned long rb\_subtree\_gap;

struct mm\_struct \*vm\_mm; //指向内存描述符

pgprot\_t vm\_page\_prot; //访问权限

unsigned long vm\_flags; //标志

union {

struct {

struct rb\_node rb;

unsigned long rb\_subtree\_last;

} linear;

struct list\_head nonlinear;

const char \_\_user \*anon\_name;

} shared;

struct list\_head anon\_vma\_chain; //匿名VMA对象链表

struct anon\_vma \*anon\_vma; //匿名VMA对象

const struct vm\_operations\_struct \*vm\_ops; //相关的操作表

unsigned long vm\_pgoff; //文件中的偏移量

struct file \* vm\_file; //被映射的文件

void \* vm\_private\_data; //私有数据

#ifndef CONFIG\_MMU

struct vm\_region \*vm\_region; /\* NOMMU mapping region \*/

#endif

};

在同一个地址空间内的不同内存区间不能重叠。

struct vm\_operations\_struct {

void (\*open)(struct vm\_area\_struct \* area);

当指定的内存区域被加入到一个地址空间时，该函数被调用

void (\*close)(struct vm\_area\_struct \* area);

当指定的内存区域从地址空间删除时，该函数被调用

int (\*fault)(struct vm\_area\_struct \*vma, struct vm\_fault \*vmf);

当没有出现在物理内存中的页面被访问时，该函数被页面故障处理调用

int (\*page\_mkwrite)(struct vm\_area\_struct \*vma, struct vm\_fault \*vmf);

当某个页面为只读页面时，该函数被页面故障处理调用

int (\*access)(struct vm\_area\_struct \*vma, unsigned long addr,

void \*buf, int len, int write);

当get\_user\_pages()函数调用失败时，该函数被access\_process\_vm()函数调用

int (\*remap\_pages)(struct vm\_area\_struct \*vma, unsigned long addr,

unsigned long size, pgoff\_t pgoff);

};

### （2）相关操作

内核时常需要在某个内存区域上执行一些操作，比如某个指定地址是否包含在某个内存区域中。这类操作非常频繁，也是mmap例程的基础。为了方便执行这类对内存区域的操作，内核定义了许多的辅助函数。

1. find\_vma

struct vm\_area\_struct \*find\_vma(struct mm\_struct \*mm, unsigned long addr)

为了找到一个给定的内存地址属于哪一个内存区域。该函数在指定的地址空间搜索第一个vm\_end大于addr的内存区域。如果没有发现这样的区域，返回NULL，否则指向匹配的内存区域vm\_area\_struct结构体指针。返回的结果被缓存在内存描述符mmap\_cache域中，有相当好的命中率，提高速度。该函数通过红黑树查找。

B．find\_vma\_prev

和find\_vma类似，但是它返回第一个小于addr的VMA。

C．find\_vma\_intersection

返回第一个和指定地址区间相交的VMA，也是调用find\_vma

D．mmap

创建一个新的线性地址区间。如果新创建的VMA和已经存在的地址区间相邻，并且相同的访问权限，两个区域将合并为一个；如果不能合并就算是一个新的VMA。创建之后，会将该地址空间加入到进程的地址空间中。

mmap系统调用对应的内核函数do\_mmap\_pgoff：

unsigned long do\_mmap\_pgoff(struct file \*file, unsigned long addr,

unsigned long len, unsigned long prot,

unsigned long flags, unsigned long pgoff,

unsigned long \*populate)

file为NULL且pgoff为0，代表没有和文件相关，称为匿名映射

如果指定了文件名和偏移量，称为文件映射

E．munmap

从特定的进程地址空间中删除指定地址空间，munmap系统调用对应的内核函数do\_munmap：

int do\_munmap(struct mm\_struct \*mm, unsigned long start, size\_t len)

## 4、缺页异常处理

Linux缺页异常处理程序必须区分两种情况：

由编程错误引起的异常

由引用属于进程地址空间但还尚未分配物理页框的页所引起的异常。

一般有如下异常及动作：

线性区有效但页面没有分配：通过物理页面分配器分配一个页面帧

线性区无效但是可以扩展（如堆栈）：扩展线性区并分配一页

页面被交换但是在交换高速缓存中：从交换高速缓存中删除并分配给进程

页面被交换至后援存储器：通过PTE中的信息查找页面并从磁盘读到内存

写只读页面：如果是COW则复制一页，标志为可写并分配给进程；否则发送SIGSEGV信号

线性区无效或者没有访问权限：发送SIGSEGV信号

异常发生在内核地址空间：如果异常发生在vmalloc就更新当前进程页表，唯一的有效发生在内核页面异常的情况

异常发生在内核态用户空间：表明内核不能从用户空间正确的复制数据，这是非常严重的内核bug。

总体方案如下图：



处理缺页中断函数（arch/arm/mm/fault.c）：

do\_translation\_fault –> do\_page\_fault -> \_\_do\_page\_fault -> handle\_mm\_fault，调用关系如下图：



pmd\_alloc/pte\_alloc：申请页表

do\_no\_page：请求页面的分配

do\_swap\_page：如果该页面被交换到磁盘则请求换页

do\_wp\_page：如果页面是写时复制，则进行保护设置

## 5、关联图

进程、线程之间内存描述符关系

进程A和B是独立进程，进程B由A创建，T1是进程A的一个线程。上图展示说明如下：

A．进程A fork进程B，进程B会复制父进程A的task\_struct和mm\_struct，进行相应的初始化，有各自的页表目录。

B．进程A pthread\_create 线程T1，复制父进程A的task\_struct，但是共享mm\_struct，使用同一个页表目录

C．进程A和进程B的虚拟地址通过mm\_struct的pgd页表映射到实际物理内存，对于可读页（如C库等），两个进程是共享实际内存，对于可写页（如堆栈、变量等）则有各自的物理内存

D．进程A和线程T1共享mm\_struct，因此也共享pgd页表，当线程需要自己的可写页时，就会加入到进程A的共享页表里。



进程的虚拟内存管理

VMA：虚拟内存区域

mm\_rb：为红黑树，图中用链表示意，表示mmap和mm\_rb都指向虚拟内存区域

每个进程都有一个task\_struct，所有进程通过链表连接起来，链表头为init\_task。每个task\_struct又都有一个mm\_struct结构，所有的mm\_struct通过mmlist连接在一个双向链表中，该链表的首元素是init\_mm。每个进程的各个虚拟内存区域通过mmap和mm\_rb组织起来，然后通过pgd页表，映射到实际的物理内存。

## 6、操作函数说明

（1）fork

创建一个具有新地址空间的进程。父进程拥有的物理页面都标记上写时复制，并且在缺页中断以前一直由两个进程所共享。一旦对现有页面发生写操作时就会产生缺页中断，从而复制该页面到新页面。

（2）clone

当设置CLONE\_VM时允许创建一个共享父进程上下文给新进程，这就是线程的实现方式。如果没有设置，这和fork一样。

（3）mmap

mmap在进程线性地址空间中创建一个新区域

（4）mremap

重映射一个内存区域或者重设其大小

（5）munmap

销毁部分或所有的区域

（6）execve

载入一个新的可执行文件，并替换掉当前的地址空间

# 四、进程间通信

常见的方法:

无名管道：只能在有血缘进程间通信，有些进程向管道里写入数据，另一些进程则从管道中读出数据

有名管道（或FIFO）：和无名管道类似，但是无血缘关系也可以通信

信号量：一般用于进程间同步

消息：允许进程在预定义的消息队列中读和写消息来交换消息（小数据块）

共享内存区：允许进程通过共享内存块来交换信息（大数据块）

套接字：允许不同计算机上的进程通过网络交换，也可以用作相同主机上的进程之间的通信工具。

# 附录A

1、参考网站

内核堆栈和用户堆栈 小结

http://www.cnblogs.com/dormant/p/5456491.html

内核态到用户态切换分析（一）

http://blog.csdn.net/armmfc/article/details/51420543

内核态到用户态切换（二）

http://blog.csdn.net/armmfc/article/details/51475569

ARM Linux上的系统调用代码分析

http://www.wowotech.net/process\_management/syscall-arm.html

O(n)、O(1)和CFS调度器

<http://www.wowotech.net/process_management/scheduler-history.html>

进程描述符task\_struct

http://www.cnblogs.com/qingjiaowoxiaoxioashou/p/5547260.html

ARM寄存器详解

http://blog.csdn.net/sandeldeng/article/details/52954781

http://liwei.life/2016/04/07/linux%E7%9A%84%E8%BF%9B%E7%A8%8B%E4%BC%98%E5%85%88%E7%BA%A7/

2、进程调度策略测试

测试原程序来自网站：<https://www.cnblogs.com/xiaotlili/p/3510224.html>，进行了修改得到。

/\* 测试结论:

\* thread1/thread2: SCHED\_RR or SCHED\_FIFO, thread3: normal

\* thread1实时优先级高于thread2, CPU个数4核

\* CPU\_NUM用于分别创建thread1 thread2 thread3个数

\* 当CPU\_NUM=1时, thread1 thread2 thread3 在3个核上运行，1个核空闲

\* 当CPU\_NUM=2时, thread1 thread2分别占据2个核运行,偶尔运行thread3

\* 当CPU\_NUM=3时, thread1占据3个核, thread2占据1个核, 偶尔运行thread3

\* 当CPU\_NUM>=4时, thread1 占据4个核, thread2得不到运行, 偶尔运行thread3

\* 问题: 为啥thread3会得到运行？

\* 答：

\* pick\_next\_task里:先执行了CFS调度类查找,再按照运行队列本身的调度类进行查找

\* \*/

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdint.h>

#include <pthread.h>

#define CPU\_NUM 4

#define LOOP\_FOREVER

void mdelay(uint32\_t ms)

{

uint32\_t i,j;

for(i=0; i<ms; i++){

for(j=0; j<100000; j++){

;

}

}

}

void Thread1()

{

sleep(1);

int i,j;

int policy;

struct sched\_param param;

pthread\_getschedparam(pthread\_self(),&policy,&param);

if(policy == SCHED\_OTHER)

printf("SCHED\_OTHER 1\n");

if(policy == SCHED\_RR);

printf("SCHED\_RR 1 \n");

if(policy==SCHED\_FIFO)

printf("SCHED\_FIFO 1\n");

#ifdef LOOP\_FOREVER

for(;;)

#else

for(i=1;i<5;i++)

#endif

{

mdelay(1000);

printf("thread 1\n");

}

printf("Pthread 1 exit\n");

}

void Thread2()

{

sleep(1);

int i,j,m;

int policy;

struct sched\_param param;

pthread\_getschedparam(pthread\_self(),&policy,&param);

if(policy == SCHED\_OTHER)

printf("SCHED\_OTHER 2\n");

if(policy == SCHED\_RR);

printf("SCHED\_RR 2\n");

if(policy==SCHED\_FIFO)

printf("SCHED\_FIFO 2\n");

#ifdef LOOP\_FOREVER

for(;;)

#else

for(i=1;i<5;i++)

#endif

{

mdelay(1000);

printf("thread 2\n");

}

printf("Pthread 2 exit\n");

}

void Thread3()

{

sleep(1);

int i,j;

int policy;

struct sched\_param param;

pthread\_getschedparam(pthread\_self(),&policy,&param);

if(policy == SCHED\_OTHER)

printf("SCHED\_OTHER 3\n");

if(policy == SCHED\_RR)

printf("SCHED\_RR 3\n");

if(policy==SCHED\_FIFO)

printf("SCHED\_FIFO 3\n");

#ifdef LOOP\_FOREVER

for(;;)

#else

for(i=1;i<5;i++)

#endif

{

mdelay(1000);

printf("thread 3\n");

}

printf("Pthread 3 exit\n");

}

int main()

{

int i;

i = getuid();

if(i==0)

printf("The current user is root\n");

else

printf("The current user is not root\n");

pthread\_t ppid1[CPU\_NUM];

pthread\_t ppid2[CPU\_NUM];

pthread\_t ppid3[CPU\_NUM];

struct sched\_param param;

pthread\_attr\_t attr,attr1,attr2;

pthread\_attr\_init(&attr1);

pthread\_attr\_init(&attr);

pthread\_attr\_init(&attr2);

param.sched\_priority = 51;

pthread\_attr\_setschedpolicy(&attr2,SCHED\_RR);

pthread\_attr\_setschedparam(&attr2,&param);

pthread\_attr\_setinheritsched(&attr2,PTHREAD\_EXPLICIT\_SCHED);//要使优先级其作用必须要有这句话

param.sched\_priority = 21;

pthread\_attr\_setschedpolicy(&attr1,SCHED\_RR);

pthread\_attr\_setschedparam(&attr1,&param);

pthread\_attr\_setinheritsched(&attr1,PTHREAD\_EXPLICIT\_SCHED);

for(i=0; i<CPU\_NUM; i++)

{

pthread\_create(&ppid3[i],&attr,(void \*)Thread3,NULL);

pthread\_create(&ppid2[i],&attr1,(void \*)Thread2,NULL);

pthread\_create(&ppid1[i],&attr2,(void \*)Thread1,NULL);

}

for(i=0; i<CPU\_NUM; i++)

{

pthread\_join(ppid3[i],NULL);

pthread\_join(ppid2[i],NULL);

pthread\_join(ppid1[i],NULL);

}

pthread\_attr\_destroy(&attr2);

pthread\_attr\_destroy(&attr1);

return 0;

}