## 进程管理的核心功能及相应原理

通过调研相关文献资料（来源于读书报告\网络博客\相关书籍），我将进程管理的核心功能大致分为四类:控制，同步，通信，调度。

进程控制:进程控制包括对单个进程本身的一些操作，比如控制进程的创建和删除以

及状态的更迭等。下面详细描述一下几个重要操作

### 1）进程的创建

一个进程可以创建一个子进程，子进程会继承父进程所拥有的资源，如继承父进程打开的文件、分配到的缓冲区等，当子进程被撤销时，应该讲其从父进程哪里获得的资源归还给父进程，此外，撤销父进程时，也必须同时撤销其所有的子进程。

进程创建的步骤包括：① 申请空白PCB，获得唯一的数字标识符，从PCB集合中索取一个空白的PCB。② 为新进程分配资源。③ 初始化进程控制块，包括：将系统分配的标识符和父进程标识符填入新的PCB中；使程序计数器指向程序的入口地址，使栈指针指向栈顶；将进程的状态设置为就绪状态或静止就绪状态。④ 将新进程插入到就绪队列，如果进程就绪队列能够接纳新进程，便将新进程插入就绪队列。

### 2） 进程的终止

引起进程终止的事件有：①正常结束②异常结束，在进程运行期间，由于出现某些错误和故障而迫使进程终止。如越界错误，非法指令，等待超时，算术运算错，I/O故障等等。③ 外界干预，进程应外界的请求而终止运行。

终止进程的步骤如下：① 根据被终止的进程的标识符，从PCB集合汇总检索除该进程的PCB，从中读出该进程的状态。② 若被终止的进程正处于执行状态，应立即终止该进程的执行，并置调度标志位真，用于指示该进程被终止后应重新进行调度。③若该进程还有子孙进程，还应将其子孙进程予以终止，以防他们成为不可控的进程。④ 将被终止的进程所拥有的全部资源，或者归还给其父进程，或者归还给操作系统。⑤ 将被终止的进程PCB从所在队列或链表中移出，等待其他程序来搜集信息。

### 3） 进程的阻塞与唤醒

引起进程阻塞与唤醒的事件有：① 请求系统服务，由于某种原因，操作系统并不立即满足该进程的要求，该进程只能转变为阻塞状态来等待。②当进程启动某种操作后，该进程必须在该操作完成之后才能继续执行，则必须先使该进程阻塞。③新数据尚未到达，进程所需数据尚未到达，该进程只有（等待）阻塞。④ 无新工作可做，系统往往设置一些具有某些特定功能的系统进程，每当这种进程完成任务后，便把自己阻塞起来以等待新任务到来。

进程阻塞步骤包括：进程的阻塞是进程自身的一种主动行为，之后进程会停止执行，并将进程的状态由执行改为阻塞，并将PCB插入阻塞队列，如果系统中设置了因不同事件而阻塞的多个阻塞队列，则应将本进程插入到具有相同事件的阻塞队列中，最后，转调度程序进行重新调度，将处理机分配给另一就绪进程并进行切换。即保留被阻塞进程的处理机状态到PCB中，再按新进程的PCB中的处理机状态设置CPU环境。

进程唤醒步骤如下：当被阻塞进程所期待的时间出现时，如I/O完成获其所期待的数据已经到达，则由有关进程（如用完并释放I/O设备的进程）调用唤醒原语wakeup，将等待该事件的进程唤醒，首先将被阻塞的进程从等待该事件的阻塞队列中移出，将其PCB中的现行状态由阻塞改为就绪，然后再将该PCB插入到就绪队列中。值得注意的是，block原语与wakeup原因应该在不同进程中执行。

进程同步：进程同步主要是对多个相关进程在执行次序上进行协调，以使并发执行的诸进程之间能有效共享资源和相互合作，而从使程序的执行具有可再现性。

总体来说，进程同步通过临界区来实现，临界区问题必须确保：互斥，前进，有限等待。临界区前后分别由进入区和退出区，其它为剩余区。

临界区问题有两类处理方法：抢占内核和非抢占内核。非抢占内核数据从根本上不会导致竞争条件，而抢占内核情况下则复杂得多。但是考虑到后者在响应时间和实时编程的优势，这种复杂值得花费力气解决。这里讨论一些solution proposals。（a）使临界区不可被打断，与非抢占内核类似，进程在临界区内时不允许上下文切换，我们可以通过一个系统调用来实现这个需求。（b）严格轮换，但是忙等会浪费CPU资源。而且轮换如此严格，使连续多次执行某个进程的临界区成为不可能。（c）对严格轮换进行改进得到Peterson’s solution，使用了两个共享数据项int turn和boolean flag。但是同样会导致忙等，而且可能会使进程的优先权错位（d）在硬件上实现互斥锁，同样忙等。

实际最终我们选择的方案是——信号量。信号量是一种数据类型，只能通过两个标准原子操作访问wait()和signal()。信号量通常分为计数信号量和二进制信号量，后者有时称为互斥锁。可以使用二进制信号量处理多进程的临界区问题，而计数信号量可以用来控制访问具有若干实例的某种资源，此时信号量表示可用资源的数量。

当一个进程位于临界区时，其他试图进入临界区的进程必须在进入代码中连续地循环，这种称为自旋锁，会导致忙等。为了克服这一点可以修改wait()和signal()地定义——当一个进程执行wait()需要等待的时候，改为阻塞自己而不是忙等。阻塞操作将此进程放入到与信号量相关的等待队列中，状态改为等待，然后会选择另一个进程来执行。

考虑进程同步时，很重要的一点是避免死锁。死锁的特征包括：互斥、占有并等待、非抢占、循环等待。当死锁发生时，进程永远不能完成，所以必须解决。有三种方法：（1）使用协议确保死锁不发生（2）允许死锁然后检测恢复（3）认为死锁不存在。

此外，进程同步中有三个经典问题，用来检验新的同步方案——生产者消费者问题、读者-写者问题、哲学家进餐问题，可以用来分析中的各种情况包括死锁问题，这里不作具体分析。

进程通信：指进程之间的信息交换，进程的互斥和同步，由于只能交换很少量的信息而被归结为低级通信，目前的高级通信机制可归结为三大类

#### ①    共享存储器系统

相互通信的进程共享某些数据结构或共享存储区，进程之间能够通过这些空间进行通信，基于此，又可以分为如下两种类型：基于共享数据结构的通信方式，在这种通信中，要求诸进程共用某些数据结构，借此实现进程间的信息交换。基于共享存储区的通信方式，为了传输大量数据，在存储器中划出一块共享存储区，诸进程可通过对共享存储区中的数据的读或写来实现通信。

#### ②    消息传递系统

进程间的数据交换是以格式化的消息为单位，程序员直接利用操作系统提供的一组通信命令（原语），不仅能实现大量数据的传递，而且还隐藏了通信的实现细节，使通信过程对用户是透明的，从而大幅减少通信程序编制的复杂性。

#### ③    管道通信

连接一个读进程和一个写进程以实现它们之间通信的一个共享文件，又名pipe文件，向管道（共享文件）提供输入的发送进程，以字符流形式将大量的数据送入管道；而接受管道输出的接受进程，则从管道中接受数据，由于发送和接受进程是利用管道进行通信的，因此叫做管道通信。管道通信需要具有三方面的协调能力：互斥（当一个进程正在对pipe执行读/写时，其他进程必须等待），同步（当写进程把一定数量的数据写入pipe，便去睡眠等待，到读进程取走数据后，再把它唤醒，当读进程读一个空pipe时，也应该睡眠等待，直到有数据写入管道，才将其唤醒），确定对方是否存在，只有确定了对方已存在时，才能进行通信。

进程调度：在多道程序系统中，进程的数量往往多于处理机的个数，进程争用处理机的情况就在所难免。处理机调度是对处理机进行分配，就是从就绪队列中，按照一定的算法（公平、髙效）选择一个进程并将处理机分配给它运行，以实现进程并发地执行。进程调度的算法多样，常见的有：

1. 先到先服务算法：即先请求cpu 的进程先分配到进程，实现简单，但平均等待时间通常较长。考虑FCFS 调度在动态情况下，会产生护航效果，会导致cpu 和设备使用率变得很低。
2. 最短作业优先调度：cpu 空闲时，它会赋给具有最短cpu 区间的进程，SJF 算法可证为最佳，其平均等待时间最小。但是困难在于如何知道下一个cpu 区间的长度。一种方法是近似SJF 调度，可以用以前cpu 长度的指数平均来预测下一个区间长度。  该算法可改善平均周转时间和平均带权周转时间，缩短进程的等待时间，提高系统的吞吐量。 但是对长进程非常不利，可能长时间得不到执行，且难以准确估计进程的执行时间，从而影响调度性能。
3. 优先级调度：每个进程都会有一个优先级，具有最高优先级的进程会被分配到cpu。这种算法的主要问题是无穷阻塞或者饥饿，可以使用老化的方法来处理
4. 轮转法调度：专门为分时系统设计，与FCFS 类似，但是增加了抢占。这里定义了一个时间片， 就绪队列作为循环队列，每个进程分配不超过一个时间片的cpu。  时间片轮转调度算法的特点是简单易行、平均响应时间短。 不利于处理紧急作业。在时间片轮转算法中，时间片的大小对系统性能的影响很大，因此时间片的大小应选择恰当
5. 多级队列调度：将就绪队列分为多个独立队列，根据进程属性每个队列有自己的调度算法。而且队列之间必须有调度，通常采用固定优先级抢占调度。例如，前台队列可以比后台队列具有绝对的优先级，这样也符合交互的要求。
6. 多级队列反馈调度：与多级队列相比，差异在于允许进程在队列之间移动

## 数据结构分析

        我们学过，进程控制块(PCB)的是进程管理的关键。一个进程是由一个进程控制块来描述的。那么首先需要做的就是找到这部分代码。在linux/sched.h中可以找到task\_struct结构体，这是一个用于进程信息保存的结构体，包含可大量的内置类型和自定义结构体指针类型，用于linux内核进程的控制能力。 下面是截取了一小部分代码：

代码较长，列举部分源码主要分析一下一下结构体内部结构变量的作用（省略号省去部分代码）：

structtask\_struct {

/\* 进程状态 \*/

volatilelongstate;

/\* 指向内核栈 \*/

void\*stack;

/\* 用于加入进程链表 \*/

structlist\_head tasks;

/\* 指向该进程的内存区描述符 \*/

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

......

/\* 进程ID，每个进程(线程)的PID都不同 \*/

pid\_t pid;

/\* 线程组ID，同一个线程组拥有相同的pid，与领头线程(该组中第一个轻量级进程)pid一致，保存在tgid中，线程组领头线程的pid和tgid相同 \*/

pid\_t tgid;

/\* 用于连接到PID、TGID、PGRP、SESSION哈希表 \*/

structpid\_link pids[PIDTYPE\_MAX];

......

/\* 指向创建其的父进程，如果其父进程不存在，则指向init进程 \*/

structtask\_struct \_\_rcu \*real\_parent;

/\* 指向当前的父进程，通常与real\_parent一致 \*/

structtask\_struct \_\_rcu \*parent;

/\* 子进程链表 \*/

structlist\_head children;

/\* 兄弟进程链表 \*/

structlist\_head sibling;

/\* 线程组领头线程指针 \*/

structtask\_struct\*group\_leader;

/\* 在进程切换时保存硬件上下文(硬件上下文一共保存在2个地方: thread\_struct(保存大部分CPU寄存器值，包括内核态堆栈栈顶地址和IO许可权限位)，内核栈(保存eax,ebx,ecx,edx等通用寄存器值)) \*/

structthread\_struct thread;

/\* 当前目录 \*/

structfs\_struct\*fs;

/\* 指向文件描述符，该进程所有打开的文件会在这里面的一个指针数组里 \*/

structfiles\_struct\*files;

......

/\*信号描述符，用于跟踪共享挂起信号队列，被属于同一线程组的所有进程共享，也就是同一线程组的线程此指针指向同一个信号描述符 \*/

structsignal\_struct\*signal;

/\*信号处理函数描述符 \*/

structsighand\_struct\*sighand;

......

}

其中关于进程的状态分为两种，struct\_task中成员state（关于运行的状态）和exit\_state（关于退出的状态），参见下图：

TASK\_RUNNING ： 这个状态是正在占有cpu或者处于就绪状态的进程才能拥有。

TASK\_INTERRUPIBLE ：进程因为等待一些条件而被挂起进（阻塞）而所处的状态。一旦等待的条件成立，进程就会从该状态（阻塞）迅速转化成为就绪状态，也就是state域的值变为TASK\_RUNNING。

TASK\_UNINTERRUPIBLE ：其实他和TASK\_INTERRUPIBLE 大致相同，除了传递一个信号和中断所引起的效果不同。

TASK\_STOP ：进程的执行被停止，当进程接收到SIGSTOP、SIGTTIN、SIGTSTP或者SIGTTOU信号之后就会进入该状态。

TASK\_TRACED ：进程执行被调试程序所停止，当一个进程被另外的进程所监视，每一个信号都会让进城进入该状态。

EXIT\_ZOMBIE ：进程已经终止，但是它的父进程还没有调用wait4或者waitpid函数来获得有关进程终止的有关信息，在调用这两个函数之前，内核不会丢弃包含死去进程的进程描述符的数据结构的，防止父进程某一个时候需要着一些信息。

EXIT\_DEAD ：进程被系统释放，因为父进程已经调用了以上所提到的函数。现在内核也就可以安全的删除该进程的一切相关无用的信息了。

## 进程创建

下面是在linux-2.6.12.1\arch\x86\_64\kernel\process.c中的系统调用代码：

按要求找到fork（）和vfork（）如下

asmlinkage long sys\_fork(structpt\_regs\*regs)

{

return do\_fork(SIGCHLD, regs->rsp, regs, 0, NULL, NULL);

}

asmlinkage long sys\_vfork(structpt\_regs\*regs)

{

return do\_fork(CLONE\_VFORK | CLONE\_VM |SIGCHLD, regs->rsp, regs, 0,

NULL, NULL);

}

       从源码可以看到do\_fork()均被上述两个系统调用所调用，但是笔者也发现do\_fork()也出现在了clone（）中，其实clone()也有创建进程的功能。则三个系统调用的执行过程如下图所示：

这里着重分析一下do\_fork();

long do\_fork(unsignedlong clone\_flags,

//该标志位的4个字节分为两部分.最低的一个字节为子进程结束时发送给父进程的信号代码,通 常为SIGCHLD(用于wait系统调用).剩余的三个字节则是各种clone标志的组合.通过clone标志可以有选择的对父进程的资源进行复制.

unsignedlong stack\_start,//未被使用,通常被赋值为0

structpt\_regs\*regs,//

unsignedlong stack\_size,

int\_\_user \*parent\_tidptr,

//父进程在用户态下pid的地址,该参数在CLONE\_PARENT\_SETTID标志被设定时有意义

int\_\_user \*child\_tidptr)

// 子进程在用户态下pid的地址,该参数在CLONE\_CHILD\_SETTID标志被设定时有意义

{

structtask\_struct\*p;

int trace =0;

long pid =alloc\_pidmap();

if (pid <0)

return-EAGAIN;

if (unlikely(current->ptrace)) {

trace = fork\_traceflag (clone\_flags);

if (trace)

clone\_flags|= CLONE\_PTRACE;

}

p = copy\_process(clone\_flags, stack\_start, regs, stack\_size,parent\_tidptr, child\_tidptr, pid)；//得到一个进程描述符, 内核栈, thread\_info与父进程一样的子进程描述符

/\*

\* Do this prior waking up the newthread - the thread pointer

\* might get invalid after thatpoint, if the thread exits quickly.

\*/

//若copy\_process执行不成功，则先释放已分配的pid，根据ptr\_err得到错误码，保存在pid中。如果成功，则执行如下代码：首先定义了一个完成量vfork，如果clone\_flags包含CLONE\_VFORK标志，那么将进程描述符中的vfork\_done字段指向这个完成量，之后再对vfork完成量进行初始化。

if (!IS\_ERR(p)){

structcompletion vfork;//定义了一个完成变量, 以便clone\_flags标志设置了CLONE\_VFORK时, 父进程阻塞, 直到子进程调用exec或exit时

if (clone\_flags & CLONE\_VFORK) {

p->vfork\_done =&vfork;

init\_completion(&vfork); //初始化完成变量

}

//如果子进程被跟踪（在父进程被跟踪的前提下，一般来说父进程是很少被跟踪的）或者设置了CLONE\_STOPPED标志，就为子进程增加挂起信号。

if ((p->ptrace & PT\_PTRACED) || (clone\_flags & CLONE\_STOPPED)) {

/\*

\* We'll start up with an immediate SIGSTOP.

\*/

sigaddset(&p->pending.signal, SIGSTOP);

set\_tsk\_thread\_flag(p,TIF\_SIGPENDING);

}

// 如果子进程未设置CLONE\_STOPPED标志，那么通过wake\_up\_new\_task函数使得父子进程之一优先运行；否则，将子进程的状态设置为TASK\_STOPPED。

if (!(clone\_flags & CLONE\_STOPPED))

wake\_up\_new\_task(p,clone\_flags); //将子进程加入红黑树,并判断子进程是否可以抢占父进程, 此时子进程已经处于运行状态

else

p->state = TASK\_STOPPED;

//如果父进程被跟踪，则将子进程的pid赋值给父进程，存于进程描述符的pstrace\_message字段。使得当前进程停止运行，并向父进程的父进程发送SIGCHLD信号。

if (unlikely (trace)) {

current->ptrace\_message = pid;

ptrace\_notify((trace <<8) |SIGTRAP);

}

//如果CLONE\_VFORK标志被设置，则通过wait操作将父进程阻塞，直至子进程调用exec函数或者退出。（这儿显示的就是vfork的作用）

if (clone\_flags & CLONE\_VFORK) {

wait\_for\_completion(&vfork);

if (unlikely (current->ptrace &PT\_TRACE\_VFORK\_DONE))

ptrace\_notify((PTRACE\_EVENT\_VFORK\_DONE <<8)|SIGTRAP);

}

} else {

free\_pidmap(pid);

pid = PTR\_ERR(p);

}

return pid; //返回子进程的进程描述符

}

vfork()与fork()的区别：

（1）vfork产生的子进程和父进程完全共享地址空间，包括代码段+数据段+堆栈段。子进程对共享资源进行的修改，也会影响到父进程。

（2）vfork函数产生的子进程一定比父进程先运行。即父进程调用了vfork函数后会等待子进程运行后再运行。

## 进程调度

首先在sched.c中发现有个注释如下:

/\*

\* schedule() is the main scheduler function.

\*/

  Ok，很显然，我们要重点研究这个函数schedule()。

  函数代码如下：

asmlinkage void \_\_sched schedule(void)

{

structtask\_struct\*prev,\*next;

structprio\_array\*array;

structlist\_head\*queue;

unsignedlonglong now;

unsignedlongrun\_time;

int cpu, idx, new\_prio;

long\*switch\_count;

structrq\*rq;

if (unlikely(in\_atomic() &&!current->exit\_state)) {

printk(KERN\_ERR "BUG: scheduling while atomic: "

"%s/0x%08x/%d/n",

current->comm, preempt\_count(), current->pid);

dump\_stack();

}

profile\_hit(SCHED\_PROFILING,\_\_builtin\_return\_address(0));

need\_resched: //禁用内核抢占并初始化一些局部变量

preempt\_disable();

prev = current;

release\_kernel\_lock(prev);

need\_resched\_nonpreemptible:

rq = this\_rq();

if (unlikely(prev == rq->idle)&& prev->state != TASK\_RUNNING) {

printk(KERN\_ERR "bad: scheduling from the idle thread!/n");

dump\_stack();

}

schedstat\_inc(rq, sched\_cnt);

spin\_lock\_irq(&rq->lock); //关掉本地中断，并获得所要保护的运行队列的自旋锁

now = sched\_clock();//调用sched\_clock( )函数以读取TSC，并将它的值转换成纳秒，所获得的时间戳存放在局部变量now中

if (likely((longlong)(now- prev->timestamp) < NS\_MAX\_SLEEP\_AVG)) {

run\_time = now -prev->timestamp;

if (unlikely((longlong)(now- prev->timestamp) <0))

run\_time =0;

} else

run\_time = NS\_MAX\_SLEEP\_AVG;

run\_time /= (CURRENT\_BONUS(prev) ?:1);

// prev可能是一个正在被终止的进程。为了确认这个事实，schedule( )检查PF\_DEAD标志：

if (unlikely(prev->flags & PF\_DEAD))

prev->state = EXIT\_DEAD;

switch\_count=&prev->nivcsw;

//检查prev的状态，如果不是可运行状态，而且它没有在内核态被抢占，就应该从运行队列删除prev进程。不过，如果它是非阻塞挂起信号，而且状态为TASK\_INTERRUPTIBLE，函数就把该进程的状态设置为TASK\_RUNNING，并将它插入运行队列：

if (prev->state &&!(preempt\_count()& PREEMPT\_ACTIVE)) {

switch\_count =&prev->nvcsw;

if (unlikely((prev->state & TASK\_INTERRUPTIBLE) &&

unlikely(signal\_pending(prev))))

prev->state = TASK\_RUNNING;

else {

if (prev->state == TASK\_UNINTERRUPTIBLE)

rq->nr\_uninterruptible++;

deactivate\_task(prev, rq);

}

}

update\_cpu\_clock(prev, rq, now);

cpu = smp\_processor\_id();

if (unlikely(!rq->nr\_running)){//如果运行队列中没有可运行的进程存在，函数就调用idle\_balance( )，从另外一个运行队列迁移一些可运行进程到本地运行队列中

idle\_balance(cpu, rq);

if (!rq->nr\_running) {

next = rq->idle;

rq->expired\_timestamp =0;

wake\_sleeping\_dependent(cpu);// 如果idle\_balance( ) 没有成功地把进程迁移到本地运行队列中，schedule( )就调用wake\_sleeping\_dependent( )重新调度空闲CPU（即每个运行swapper进程的CPU）中的可运行进程。

goto switch\_tasks;

}

}

//假设schedule( )函数已经肯定运行队列中有一些可运行的进程，现在它必须检查这些可运行进程中是否至少有一个进程是活动的，如果没有，函数就交换运行队列数据结构的active和expired字段的内容，因此，所有的过期进程变为活动进程，而空集合准备接纳将要过期的进程。

array = rq->active;

if (unlikely(!array->nr\_active)){

/\*

\* Switch the active and expiredarrays.

\*/

schedstat\_inc(rq, sched\_switch);

rq->active = rq->expired;

rq->expired = array;

array = rq->active;

rq->expired\_timestamp =0;

rq->best\_expired\_prio = MAX\_PRIO;

}

idx = sched\_find\_first\_bit(array->bitmap);

queue = array->queue + idx;

next = list\_entry(queue->next, structtask\_struct, run\_list);

if (!rt\_task(next)&&interactive\_sleep(next->sleep\_type)) {

unsignedlonglong delta = now - next->timestamp;

if (unlikely((longlong)(now- next->timestamp) <0))

delta =0;

if (next->sleep\_type == SLEEP\_INTERACTIVE)

delta = delta \* (ON\_RUNQUEUE\_WEIGHT \*128/100) /128;

array = next->array;

new\_prio = recalc\_task\_prio(next, next->timestamp + delta);

if (unlikely(next->prio !=new\_prio)) {//如果优先级不相等

dequeue\_task(next, array);

next->prio =new\_prio;

enqueue\_task(next, array);

}

}

next->sleep\_type = SLEEP\_NORMAL;

if (dependent\_sleeper(cpu, rq, next))

next = rq->idle;

switch\_tasks:

if (next == rq->idle)

schedstat\_inc(rq, sched\_goidle);

prefetch(next);//prefetch 宏提示CPU控制单元把next进程描述符的第一部分字段的内容装入硬件高速缓存

prefetch\_stack(next);

clear\_tsk\_need\_resched(prev);

rcu\_qsctr\_inc(task\_cpu(prev));

//减少prev的平均睡眠时间，并把它补充给进程所使用的CPU时间片

prev->sleep\_avg -= run\_time;

if ((long)prev->sleep\_avg <=0)

prev->sleep\_avg =0;

prev->timestamp = prev->last\_ran= now;

sched\_info\_switch(prev, next);

if (likely(prev != next)) {rev 和next很可能是同一个进程：在当前运行队列中没有优先权较高或相等的其他活动进程时，会发生这种情况。在这种情况下，函数不做进程切换

next->timestamp = now;

rq->nr\_switches++;

rq->curr =next;

++\*switch\_count;

prepare\_task\_switch(rq, prev, next);

prev = context\_switch(rq, prev, next);

barrier();

finish\_task\_switch(this\_rq(), prev);

} else

spin\_unlock\_irq(&rq->lock);

prev = current;

if (unlikely(reacquire\_kernel\_lock(prev) <0))

goto need\_resched\_nonpreemptible;

preempt\_enable\_no\_resched();

if (unlikely(test\_thread\_flag(TIF\_NEED\_RESCHED)))

goto need\_resched;

}

unsignedlongpolicy;

关于调度策略，是在sched.h定义的三种

/\*

\* Scheduling policies

\*/

#defineSCHED\_NORMAL 0

#defineSCHED\_FIFO 1

#defineSCHED\_RR 2

SCHED\_NORMAL：普通进程使用的调度策略，现在此调度策略使用的是CFS调度器。

SCHED\_FIFO：实时进程使用的调度策略，此调度策略的进程一旦使用CPU则一直运行，直到有比其更高优先级的实时进程进入队列，或者其自动放弃CPU，适用于时间性要求比较高，但每次运行时间比较短的进程。

SCHED\_RR：实时进程使用的时间片轮转法策略，实时进程的时间片用完后，调度器将其放到队列末尾，这样每个实时进程都可以执行一段时间。适用于每次运行时间比较长的实时进程。

具体方法可以在sched.c中看到——sched\_setscheduler()函数将pid所指定进程的调度策略和调度参数分别设置为param指向的sched\_param结构中指定的policy和参数。

sched\_param结构中的sched\_priority成员的值可以为任何整数，该整数位于policy所指定调度策略的优先级范围内(含边界值)。

## 其他收获

首先这次选用vscode作为工具进行阅读源码，相当有效，主要在于可以很容易地Goto Declaretion,尤其源码繁杂调用较多的情况下，解决了函数或者变量找不到定义地方的问题，用起来十分方便。

但是仍会有多个定义不知道用哪个的情况。这个是LINUX支持的CPU较多造成的。我们可以用GDB静态分析，也可以用objdump和nm等工具来精确定位一些函数和变量，也可以根据宏来一步一步分析，比如是mips的那么我们就进mips看，这样一步一步来。

---------------------