**任务：分析进程调度算法，给出调度原理，程序运行的流程，相关的数据结构，进程的切换的流程，进程的上下文恢复和还原的 具体代码。**，

涉及到的源文件主要位于linux0.11内核代码如下

ｉｎｃｌｕｄｅ／ｌｉｎｕｘ／ｓｃｈｅｄ．ｈ

ｋｅｒｎｅｌ／ｆｏｒｋ．ｃ

ｋｅｒｎｅｌ／ｓｃｈｅｄ．ｃ

ｋｅｒｎｅｌ／ｅｘｉｔ．ｃ

ｋｅｒｎｅｌ／ｓｙｓｔｅｍ＿ｃａｌｌｓ．ｃ

**（1）分析进程调度算法给出调度原理**

进程运行需要各种各样的系统资源，调度的实质就是资源的分配。系统通过不同的调度算法（Scheduling Algorithm）来实现这种资源的分配。通常来说，选择什么样的调度算法取决于的资源分配的策略（Scheduling Policy）

主要的调度算法及其基本原理如下

1、时间片轮转调度算法

时间片（Time Slice）就是分配给进程运行的一段时间。

在分时系统中，为了保证人机交互的及时性，系统使每个进程依次地按时间片轮流的方式执行，此时即应采用时间片轮转法进行调度。在通常的轮转法中，系统将所有的可运行（即就绪）进程按先来先服务的原则，排成一个队列，每次调度时把CPU分配给队首进程，并令其执行一个时间片。时间片的大小从几ms到几百ms不等。当执行的时间片用完时，系统发出信号，通知调度程序，调度程序便据此信号来停止该进程的执行，并将它送到运行队列的末尾，等待下一次执行；然后，把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证运行队列中的所有进程，在一个给定的时间（人所能接受的等待时间）内，均能获得一时间片的处理机执行时间。

2、优先权调度算法

为了照顾到紧迫型进程在进入系统后便能获得优先处理，引入了最高优先权调度算法。当将该算法用于进程调度时，系统将把处理机分配给运行队列中优先权最高的进程，这时，又可进一步把该算法分成两种方式：

(1) 非抢占式优先权算法（又称不可剥夺调度：Nonpreemptive Scheduling）

在这种方式下，系统一旦将处理机（CPU）分配给运行队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可将处理机分配给另一个优先权高的进程。这种调度算法主要用于批处理系统中，也可用于某些对实时性要求不严的实时系统中。

(2) 抢占式优先权调度算法（又称可剥夺调度：Preemptive Scheduling）

该算法的本质就是系统中当前运行的进程**永远**是可运行进程中优先权最高的那个。

在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但是只要一出现了另一个优先权更高的进程时，调度程序就暂停原最高优先权进程的执行，而将处理机分配给新出现的优先权最高的进程，即剥夺当前进程的运行。因此，在采用这种调度算法时，每当出现一新的可运行进程，就将它和当前运行进程进行优先权比较，如果高于当前进程，将触发进程调度。

这种方式的优先权调度算法，能更好的满足紧迫进程的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。Linux也采用这种调度算法。

3、多级反馈调度算法

这是时下最时髦的一种调度算法。其本质是：综合了时间片轮转调度和抢占式优先权调度的优点，即：优先权高的进程先运行给定的时间片，相同优先权的进程轮流运行给定的时间片。

4、实时调度

最后我们来看一下实时系统中的调度。什么叫实时系统，就是系统对外部事件有求必应、尽快响应。在实时系统中存在有若干个实时进程或任务，它们用来反应或控制某个（些）外部事件，往往带有某种程度的紧迫性，因而对实时系统中的进程调度有某些特殊要求。

在实时系统中，广泛采用抢占调度方式，特别是对于那些要求严格的实时系统。因为这种调度方式既具有较大的灵活性，又能获得很小的调度延迟；但是这种调度方式也比较复杂。

**\*在Linux中采用了优先权调度算法。**

有关的函数是schedule（）由它来决定是否要进行进程的切换，如果要切换的话，切换到哪个进程等等，进程的调度依据如下：该函数首先扫描任务数组，通过比较每个就绪态任务的运行时间递减滴答计数counter的值来确定那个进程的运行时间最少，哪一个的值答，就表示运行时间还不长，于是九选中该进程，并使用任务切换宏函数切换到该进程。如果此时所有处于就绪态的进程的时间片都已经用完，系统就会根据每个进程的优先权值priority，对于系统中的所有进程（包括所有正在睡眠的进程）重新计算每个任务需要运行的时间片 值。然后schedule函数重新扫描任务数组中所有处于就绪态的进程，并重复上述过程，直到选择除一个进程为止，最后调用switch\_to（）执行实际的进程切换操作。如果此时没有其他进程可运行，系统就会选择进程0运行，在Linux0.11下进程0会调用pause（）把自己设置为可中断的睡眠状态并再次调用schedule（）。不过在调度进程运行时，schedule并不在意进程0处于什么状态。之钥系统空闲九调度进程0运行。

具体的代码位于ｋｅｒｎｅｌ／ｓｃｈｅｄ．ｃ中，具体如下

void

schedule (void)

{

int i, next, c;

struct task\_struct \*\*p; /\* 任务结构指针的指针。 \*/

/\* check alarm, wake up any interruptible tasks that have got a signal \*/

/\* 检测alarm（进程的报警定时值），唤醒任何已得到信号的可中断任务 \*/

/\* 从任务数组中最后一个任务开始检测alarm。 \*/

for (p = &LAST\_TASK; p > &FIRST\_TASK; --p)

if (\*p)

{

/\* 如果设置过任务的定时值alarm，并且已经过期(alarm〈jiffies〉，则在信号位图中置SIGALRM

信号，即向任务发送SIGALARM信号。然后清alarm。该信号的默认操作是终止进程。jiffies

是系统从开机开始算起的滴答数（10ms/滴答）。定义在sched.h第139行。 \*/

if ((\*p)->alarm && (\*p)->alarm < jiffies)

{

(\*p)->signal |= (1 << (SIGALRM - 1));

(\*p)->alarm = 0;

}

/\* 如果信号位图中除被阻塞的信号外还有其它信号，并且任务处于可中断状态，则置任务为就绪状态。 \*/

/\* 其中'~(\_BLOCKABLE & (\*p)->blocked)'用于忽略被阻塞的信号，但SIGKILL 和SIGSTOP 不能被阻塞。 \*/

if (((\*p)->signal & ~(\_BLOCKABLE & (\*p)->blocked)) &&

(\*p)->state == TASK\_INTERRUPTIBLE)

(\*p)->state = TASK\_RUNNING; /\*置为就绪（可执行）状态。 \*/

}

/\* this is the scheduler proper: \*/

/\* 这里是调度程序的主要部分 \*/

while (1)

{

c = -1;

next = 0;

i = NR\_TASKS;

p = &task[NR\_TASKS];

/\* 这段代码也是从任务数组的最后一个任务开始循环处理，并跳过不含任务的数组槽。比较每个就绪 \*/

/\* 状态任务的counter（任务运行时间的递减滴答计数）值，哪一个值大，运行时间还不长，next 就 \*/

/\* 指向哪个的任务号。 \*/

while (--i)

{

if (!\*--p)

continue;

if ((\*p)->state == TASK\_RUNNING && (\*p)->counter > c)

c = (\*p)->counter, next = i;

}

/\* 如果比较得出有counter值不等于0的结果，或者系统中没有一个可运行的任务存在（此时c

仍然为-1，neXt=0），则退出124行开始的循环，执行141行上的任务切换操作。否则就根据

每个任务的优先权值，更新每一个任务的counter值，然后回到125行重新比较。counter值

的计算方式为counter = counter /2 + priority。注意，这里计算过程不考虑进程的状态。 \*/

if (c)

break;

for (p = &LAST\_TASK; p > &FIRST\_TASK; --p)

if (\*p)

(\*p)->counter = ((\*p)->counter >> 1) + (\*p)->priority;

}

/\* 用下面宏（定义在sched.h中）把当前任务指针current指向任务号为next的任务，并切换

到该任务中运行。在126行next被初始化为0。因此若系统中没有任何其他任务可运行时，

则next始终为0。因此调度函数会在系统空闲时去执行任务0。此时任务0仅执行pause()

II系统调用，并又会调用本函数。 \*/

switch\_to (next); /\* 切换到任务号为next 的任务，并运行之。 \*/

}

int

sys\_pause (void)

{

current->state = TASK\_INTERRUPTIBLE;

schedule ();

return 0;

}

/\* 把当前任务置为不可中断的等待状态，并让睡眠队列头指针指向当前任务。

\* 只有明确地唤醒时才会返回。该函数提供了进程与中断处理程序之间的同步机制。

\* 函数参数P是等待任务队列头指针。指针是含有一个变量地址的变量。这里参数P使用了指针的

\* 指针形式’\*\*P’，这是因为C函数参数只能传值，没有直接的方式让被调用函数改变调用该函数

\* 程序中变量的值。但是指针’\*P’指向的目标（这里是任务结构）会改变，因此为了能修改调用该

\* 函数程序中原来就是指针变量的值，就需要传递指针’\*P’的指针，即’\*\*P’。参见图8-6中p指针

\* 的使用情况。

\*/

void

sleep\_on (struct task\_struct \*\*p)

{

struct task\_struct \*tmp;

/\* 若指针无效，则退出。（指针所指的对象可以是NULL，但指针本身不应该为0）。另外，如果

当前任务是任务0，则死机。因为任务0的运行不依赖自己的状态，所以内核代码把任务0置

为睡眠状态毫无意义。 \*/

if (!p)

return;

if (current == &(init\_task.task)) /\* 如果当前任务是任务0，则死机(impossible!)。 \*/

panic ("task[0] trying to sleep");

/\* 让tmp指向已经在等待队列上的任务(如果有的话)，例如inode->i—wait。并且将睡眠队列头

的等待指针指向当前任务。这样就把当前任务插入到了 \*P的等待队列中。然后将当前任务置

为不可中断的等待状态，并执行重新调度。 \*/

tmp = \*p; /\* 让tmp 指向已经在等待队列上的任务(如果有的话)。 \*/

\*p = current; /\* 将睡眠队列头的等待指针指向当前任务。 \*/

current->state = TASK\_UNINTERRUPTIBLE; /\* 将当前任务置为不可中断的等待状态。 \*/

schedule (); /\* 重新调度。 \*/

/\* 只有当这个等待任务被唤醒时，调度程序才又返回到这里，表示本进程已被明确地唤醒（就

绪态）。既然大家都在等待同样的资源，那么在资源可用时，就有必要唤醒所有等待该资源

的进程。该函数嵌套调用，也会嵌套唤醒所有等待该资源的进程。这里嵌套调用是指当一个

进程调用了 sleep\_on()后就会在该函数中被切换掉，控制权被转移到其他进程中。此时若有

进程也需要使用同一资源，那么也会使用同一个等待队列头指针作为参数调用sleep\_on()函

数，并且也会“陷入”该函数而不会返回。只有当内核某处代码以队列头指针作为参数wake\_up

了该队列，那么当系统切换去执行头指针所指的进程A时，该进程才会继续执行163行，把

队列后一个进程B置位就绪状态（唤醒）。而当轮到B进程执行时，它也才可能继续执行

163行。若它后面还有等待的进程C，那么它也会把C唤醒等。这里在163行前还应该添加

一条语句：\*P = tmp;见183行上的解释。

\*/

if (tmp) /\* 若还存在等待的任务，则也将其置为就绪状态（唤醒）。 \*/

tmp->state = 0;

**（2）进程的切换流程**

Linux内核支持多个进程虚拟地并发执行，这是通过不断地保存和切换程序的运行环境而实现的，选择哪个进程运行是由调度程序决定的.

前面所介绍的schedule()中调用了switch\_to宏,这个宏实现了进程之间的真正切换，每当选择出一个新的可运行进程时，schedule（）函数就会调用定义在include/asm/system．ｈ中的switch-to（）宏执行实际的进程切换操作，

。该宏会把cpu的当前进程状态替换成新进程的状态。在进行切换之前，switchto（）首先检查要切换到的进程是否就是当前进程，如果时，就什么也不做直接退出，如果不是，就首先把内核全局变量current设置为新任务的指针，然后长跳转到新任务的任务状态段TSS组成的地址处，造成CPU执行任务切换操作。此时CPU会把其所有寄存器的状态保存在当前任务寄存器TR中TSS段选择符所指向的当前进程任务数据结构的ｔｓｓ结构中，然后把新任务状态选择段选择符所指向的新任务数据结构中ｔｓｓ结构中的寄存器信息恢复到CPU中。系统就正式开始运行新的任务了。

具体的代码如下

#define switch\_to(n) {\

struct {long a,b;} \_\_tmp; \

/\* 任务n 是当前任务吗?(current ==task[n]?) \*/\_\_asm\_\_("cmpl %%ecx,\_current\n\t" \

/\* 是，则什么都不做，退出。 \*/ "je 1f\n\t" \

/\* 将新任务TSS的16位选择符——>\_\_tmp.b。 \*/ "movw %%dx,%1\n\t" \

/\* current = task[n]；ecx =被切换出的任务。 \*/ "xchgl %%ecx,\_current\n\t" \

/\* 执行长跳转至\*&—tmp，造成任务切换。 \*/ "ljmp %0\n\t" \

/\* 在任务切换回来后才会继续执行下面的语句。 \*/ \

/\* 原任务上次使用过协处理器吗？ \*/ "cmpl %%ecx,\_last\_task\_used\_math\n\t" \

/\* 没有则跳转，退出。 \*/ "jne 1f\n\t" \

/\* 原任务上次使用过协处理器，则清crO中的任 \*/ "clts\n" \

/\* 务切换务切换标志 TS。 \*/ "1:" \

::"m" (\*&\_\_tmp.a),"m" (\*&\_\_tmp.b), \

"d" (\_TSS(n)),"c" ((long) task[n])); \

}

/\* 页面地址对准。（在内核代码中没有任何地方引用!!） \*/

**（3）进程有关的数据结构如下**

定义在头文件include/Linux/sched.h中。具体解释如下所示

/\* 这里是任务（进程）数据结构，或称为进程描述符。

\* ==========================

struct task\_struct

{

\* long state 任务的运行状态（-1 不可运行，0 可运行(就绪)，>0 已停止）。

\* long counter 任务运行时间计数(递减)（滴答数），运行时间片。

\* long priority 运行优先数。任务开始运行时counter = priority，越大运行越长。

\* long signal 信号。是位图，每个比特位代表一种信号，信号值=位偏移值+1。

\* struct sigaction sigaction[32] 信号执行属性结构，对应信号将要执行的操作和标志信息。

\* long blocked 进程信号屏蔽码（对应信号位图）。

\* --------------------------

\* int exit\_code 任务执行停止的退出码，其父进程会取。

\* unsigned long start\_code 代码段地址。

\* unsigned long end\_code 代码长度（字节数）。

\* unsigned long end\_data 代码长度 + 数据长度（字节数）。

\* unsigned long brk 总长度（字节数）。

\* unsigned long start\_stack 堆栈段地址。

\* long pid 进程标识号(进程号)。

\* long father 父进程号。

\* long pgrp 父进程组号。

\* long session 会话号。

\* long leader 会话首领。

\* unsigned short uid 用户标识号（用户id）。

\* unsigned short euid 有效用户id。

\* unsigned short suid 保存的用户id。

\* unsigned short gid 组标识号（组id）。

\* unsigned short egid 有效组id。

\* unsigned short sgid 保存的组id。

\* long alarm 报警定时值（滴答数）。

\* long utime 用户态运行时间（滴答数）。

\* long stime 系统态运行时间（滴答数）。

\* long cutime 子进程用户态运行时间。

\* long cstime 子进程系统态运行时间。

\* long start\_time 进程开始运行时刻。

\* unsigned short used\_math 标志：是否使用了协处理器。

\* --------------------------

\* int tty 进程使用tty 的子设备号。-1 表示没有使用。

\* unsigned short umask 文件创建属性屏蔽位。

\* struct m\_inode \* pwd 当前工作目录i 节点结构。

\* struct m\_inode \* root 根目录i 节点结构。

\* struct m\_inode \* executable 执行文件i 节点结构。

\* unsigned long close\_on\_exec 执行时关闭文件句柄位图标志。（参见include/fcntl.h）

\* struct file \* filp[NR\_OPEN] 进程使用的文件表结构。

\* --------------------------

\* struct desc\_struct ldt[3] 本任务的局部表描述符。0-空，1-代码段cs，2-数据和堆栈段ds&ss。

\* --------------------------

\* struct tss\_struct tss 本进程的任务状态段信息结构。

}

\* ==========================

\*/

**（4）程序运行的流程**

程序只是一个普通文件，是一个机器代码指令和数据的集合，这些指令和数据存储在磁盘上的一个可执行映象（Executable Image）中，程序装入内存后就可以运行了：在指令指针寄存器的控制下，不断的将指令取至CPU运行。这些指令控制的对象不外乎各种存储器（内存、外存和各种CPU寄存器等），这些存储器中保存有待运行的指令和待处理的数据，当然，指令只有得到CPU才能发挥其作用。可见，在计算机内部，程序的执行过程实际上就是一个执行环境的总和，这个执行环境包括程序中各种指令和数据，还有一些额外数据，比如说寄存器的值、用来保存临时数据（例如传递给某个函数的参数、函数的返回地址、保存变量等）的堆栈（包括程序堆栈和系统堆栈）、被打开文件的数量及输入输出设备的状态等等。这个执行环境的动态变化表征程序的运行。我们就把这个环境称作“进程”，它代表程序的执行过程，是一个动态的实体，它随着程序中指令的执行而不断地变化。在某个特定时刻的进程的内容被称为进程映象（process image）。

进程执行时，它会根据具体情况改变状态 。进程状态是调度和对换的依据，状态的介绍如下

（１）**可运行状态**　　　处于这种状态的进程，要么正在运行、要么正准备运行。正在运行的进程就是当前进程（由current所指向的进程），而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。在后面我们讨论进程调度的时候，可以看到运行队列的作用。当前运行进程一直处于该队列中，也就是说，current总是指向运行队列中的某个元素，只是具体指向谁由调度程序决定。

（２）**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。Linux中处于等待状态的进程分为两种：可中断的等待状态和不可中断的等待状态。处于可中断等待态的进程可以被信号唤醒，如果收到信号，该进程就从等待状态进入可运行状态，并且加入到运行队列中，等待被调度；而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。

（３）**暂停状态**　　　　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。

（４）**僵死状态**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。

当一个进程的时间片用完时，系统就会使用调度程序强制切换到其他的进程去执行，另外进程在内核态执行时需要等待系统的某个资源，此时进程就会调用ｓｌｅｅｐ＿ｏｎ（）或interruptｉｂｌｅ＿ｓｌｅｅｐ＿ｏｎ（）自愿的放弃CPU的使用权，而让调度程序去执行其他的进程，进程则进入睡眠状态。

只有进程从内核运行态转移到睡眠状态时，内核才会进行进程的切换操作，在内核态下运行的进程不能被其他进程抢占，而且一个进程不能改变另一个进程的状态。

（１）Linux系统中创建新进程使用fork（）系统调用，所有进程都是通过复制进程０而得到的，都是进程０的子进程。相关代码位于kernel／fork中

int copy\_mem(int nr,struct task\_struct \* p)

{

unsigned long old\_data\_base,new\_data\_base,data\_limit;

unsigned long old\_code\_base,new\_code\_base,code\_limit;

/\* 首先取当前进程局部描述符表中代码段描述符和数据段描述符项中的段限长（字节数）。

0x0f是代码段选择符；0x17是数据段选择符。然后取当前进程代码段和数据段在线性地址

空间中的基地址。由于Linux0.11内核还不支持代码和数据段分立的情况，因此这里需要

检查代码段和数据段基址和限长是否都分别相同。否则内核显示出错信息，并停止运行。

get\_limit()和 get\_base()定义在 include/linux/sched.h 第 226 行处。

\*/

code\_limit=get\_limit(0x0f); /\* 取局部描述符表中代码段描述符项中段限长。 \*/

data\_limit=get\_limit(0x17); /\* 取局部描述符表中数据段描述符项中段限长。 \*/

old\_code\_base = get\_base(current->ldt[1]); /\* 取原代码段基址。 \*/

old\_data\_base = get\_base(current->ldt[2]); /\* 取原数据段基址。 \*/

if (old\_data\_base != old\_code\_base)

panic("We don't support separate I&D");

if (data\_limit < code\_limit) /\* 如果数据段长度 < 代码段长度也不对。 \*/

panic("Bad data\_limit");

/\* 然后设置创建中的新进程在线性地址空间中的基地址等于（64MB \*其任务号），并用该值

设置新进程局部描述符表中段描述符中的基地址。接着设置新进程的页目录表项和页表项，

即复制当前进程（父进程）的页目录表项和页表项。此时子进程共享父进程的内存页面。

正常情况下copy—page—tablesO返回0，否则表示出错，则释放刚申请的页表项。

\*/

new\_data\_base = new\_code\_base = nr \* 0x4000000; /\* 新基址=任务号\*64Mb(任务大小)。 \*/

p->start\_code = new\_code\_base;

set\_base(p->ldt[1],new\_code\_base); /\* 设置代码段描述符中基址域。 \*/

set\_base(p->ldt[2],new\_data\_base); /\* 设置数据段描述符中基址域。 \*/

if (copy\_page\_tables(old\_data\_base,new\_data\_base,data\_limit))

{ /\* 复制代码和数据段。 \*/

free\_page\_tables(new\_data\_base,data\_limit); /\* 如果出错则释放申请的内存。 \*/

return -ENOMEM;

}

return 0;

}

/\*

\* Ok, this is the main fork-routine. It copies the system process

\* information (task[nr]) and sets up the necessary registers. It

\* also copies the data segment in it's entirety.

\*/

/\*

\* OK，下面是主要的fork 子程序。它复制系统进程信息(task[n])并且设置必要的寄存器。

\* 它还整个地复制数据段。

\*/

/\* 复制进程。

该函数的参数是进入系统调用中断处理过程（system—call.s）开始，直到调用本系统调用处理

过程（system—call.s第208行）和调用本函数前时（system—call.s第217行）逐步压入栈的

各寄存器的值。这些在system—call.s程序中逐步压入栈的值（参数）包括：

①CPU执行中断指令压入的用户栈地址ss和esp、标志寄存器eflags和返回地址cs和eip;

②第83—88行在刚进入system—call时压入栈的段寄存器ds、es、fs和edx、ecx、ebx;

③第94行调用sys—call—table中sys—fork函数时压入栈的返回地址（用参数none表示）；

④第 212--216 行在调用 copy—process()之前压入栈的 gs、esi、edi、ebp 和 eax(nr)值。

其中参数nr是调用find—empty—process()分配的任务数组项号。

\*/

int copy\_process(int nr,long ebp,long edi,long esi,long gs,long none,

long ebx,long ecx,long edx,

long fs,long es,long ds,

long eip,long cs,long eflags,long esp,long ss)

{

struct task\_struct \*p;

int i;

struct file \*f;

/\* 首先为新任务数据结构分配内存。如果内存分配出错，则返回出错码并退出。然后将新任务

结构指针放入任务数组的nr项中。其中nr为任务号，由前面find—empty—process()返回。

接着把当前进程任务结构内容复制到刚申请到的内存页面P开始处。

\*/

p = (struct task\_struct \*) get\_free\_page(); /\* 为新任务数据结构分配内存。 \*/

if (!p) /\* 如果内存分配出错，则返回出错码并退出。 \*/

return -EAGAIN;

task[nr] = p; /\* 将新任务结构指针放入任务数组中。 \*/

/\* 其中nr 为任务号，由前面find\_empty\_process()返回。 \*/

\*p = \*current; /\* NOTE! this doesn't copy the supervisor stack \*/

/\* 注意！这样做不会复制超级用户的堆栈 （只复制当前进程内容）。 \*/

/\* 随后对复制来的进程结构内容进行一些修改，作为新进程的任务结构。先将新进程的状态

置为不可中断等待状态，以防止内核调度其执行。然后设置新进程的进程号pid和父进程

号father，并初始化进程运行时间片值等于其priority值（一般为15个嘀塔）。接着

复位新进程的信号位图、报警定时值、会话（session）领导标志leader、进程及其子

进程在内核和用户态运行时间统计值，还设置进程开始运行的系统时间start—time。

\*/

p->state = TASK\_UNINTERRUPTIBLE;

p->pid = last\_pid; /\* 新进程号。由前面调用find\_empty\_process()得到。 \*/

p->father = current->pid; /\* 设置父进程号。 \*/

p->counter = p->priority; /\* 运行时间片值。 \*/

p->signal = 0; /\* 信号位图置0。 \*/

p->alarm = 0; /\* 报警定时值（滴答数）。 \*/

p->leader = 0; /\* process leadership doesn't inherit \*/

/\* 进程的领导权是不能继承的 \*/

p->utime = p->stime = 0; /\* 初始化用户态时间和核心态时间。 \*/

p->cutime = p->cstime = 0; /\* 初始化子进程用户态和核心态时间。 \*/

p->start\_time = jiffies; /\* 进程开始运行时间（当前时间滴答数）。 \*/

/\* 再修改任务状态段TSS数据（参见列表后说明）。由于系统给任务结构p分配了 1页新

内存，所以（PAGE—SIZE + (long) p）让espO正好指向该页顶端。ss0:esp0用作程序

在内核态执行时的栈。另外，在第3章中我们已经知道，每个任务在GDT表中都有两个

段描述符，一个是任务的TSS段描述符，另一个是任务的LDT表段描述符。下面111行

语句就是把GDT中本任务LDT段描述符的选择符保存在本任务的TSS段中。当CPU执行

切换任务时，会自动从TSS中把LDT段描述符的选择符加载到ldtr寄存器中。

\*/

p->tss.back\_link = 0; /\* 以下设置任务状态段TSS 所需的数据（参见列表后说明）。 \*/

p->tss.esp0 = PAGE\_SIZE + (long) p; /\* 任务内核态栈指针。堆栈指针（由于是给任务结构p 分配 \*/

/\* 了1 页新内存，所以此时esp0 正好指向该页顶端）。 \*/

p->tss.ss0 = 0x10; /\* 内核态栈的段选择符（与内核数据段相同）。 \*/

p->tss.eip = eip; /\* 指令代码指针。 \*/

p->tss.eflags = eflags; /\* 标志寄存器。 \*/

p->tss.eax = 0; /\* 这是当fork()返回时新进程会返回0的原因所在。 \*/

p->tss.ecx = ecx;

p->tss.edx = edx;

p->tss.ebx = ebx;

p->tss.esp = esp;

p->tss.ebp = ebp;

p->tss.esi = esi;

p->tss.edi = edi;

p->tss.es = es & 0xffff; /\* 段寄存器仅16 位有效。 \*/

p->tss.cs = cs & 0xffff;

p->tss.ss = ss & 0xffff;

p->tss.ds = ds & 0xffff;

p->tss.fs = fs & 0xffff;

p->tss.gs = gs & 0xffff;

p->tss.ldt = \_LDT(nr); /\* 该新任务nr 的局部描述符表选择符（LDT 的描述符在GDT 中）。 \*/

p->tss.trace\_bitmap = 0x80000000; /\* (高 16 位有效)。 \*/

/\* 如果当前任务使用了协处理器，就保存其上下文。汇编指令cits用于清除控制寄存器CR0

中的任务已交换（TS）标志。每当发生任务切换，CPU都会设置该标志。该标志用于管理

数学协处理器：如果该标志置位，那么每个ESC指令都会被捕获（异常7）。如果协处理

器存在标志MP也同时置位的话，那么WAIT指令也会捕获。因此，如果任务切换发生在一

个ESC指令开始执行之后，则协处理器中的内容就可能需要在执行新的ESC指令之前保存

起来。捕获处理句柄会保存协处理器的内容并复位TS标志。指令fnsave用于把协处理器

的所有状态保存到目的操作数指定的内存区域中（tss.i387）。

\*/

if (last\_task\_used\_math == current)

\_\_asm\_\_("clts ; fnsave %0"::"m" (p->tss.i387));

/\* 接下来复制进程页表。即在线性地址空间中设置新任务代码段和数据段描述符中的基址

和限长，并复制页表。如果出错（返回值不是0），则复位任务数组中相应项并释放为

该新任务分配的用于任务结构的内存页。

\*/

if (copy\_mem(nr,p))

{ /\* 返回不为0 表示出错。 \*/

task[nr] = NULL;

free\_page((long) p);

return -EAGAIN;

}

/\* 如果父进程中有文件是打开的，则将对应文件的打开次数增1。因为这里创建的子进程

会与父进程共享这些打开的文件。将当前进程（父进程）的pwd，root和executable

引用次数均增1。与上面同样的道理，子进程也引用了这些i节点。

\*/

for (i=0; i<NR\_OPEN;i++)

if (f=p->filp[i])

f->f\_count++;

/\* 将当前进程（父进程）的pwd, root 和executable 引用次数均增1。 \*/

if (current->pwd)

current->pwd->i\_count++;

if (current->root)

current->root->i\_count++;

if (current->executable)

current->executable->i\_count++;

/\* 随后在GDT表中设置新任务TSS段和LDT段描述符项。这两个段的限长均被设置成104字

节。set—tss—desc()和set—ldt—desc()的定义参见 include/asm/system.h 文

件52—66行和码。“gdt+(nr?l)+FIRST—TSS—ENTRY”是任务nr的TSS描述符项在全局

表中的地址。因为每个任务占用GDT表中2项，因此上式中要包括’（nr?l）’。

程序然后把新进程设置成就绪态。另外在任务切换时，任务寄存器tr由CPU自动加载。

最后返回新进程号。

\*/

set\_tss\_desc(gdt+(nr<<1)+FIRST\_TSS\_ENTRY,&(p->tss));

set\_ldt\_desc(gdt+(nr<<1)+FIRST\_LDT\_ENTRY,&(p->ldt));

p->state = TASK\_RUNNING; /\* do this last, just in case \*/

/\* 最后再将新任务设置成可运行状态，以防万一 \*/

return last\_pid; /\* 返回新进程号（与任务号是不同的）。 \*/

}

（２）通过调度算法对进程进行调度和切换，具体规则见上。

（３）进程结束运行活在半途停止时，内核需要释放该进程所占用的系统资源。包括进程运行时打开的文件、申请的内存等。

当一个用户程序调用exit（）系统调用时，就会执行内核函数ｄｏ＿ｅｘｉｔ（）相关代码位于kernel／exit中

int

do\_exit (long code) /\* code 是错误码。 \*/

{

int i;

/\* 首先释放当前进程代码段和数据段所占的内存页。函数free—page—tablesO的第1个参数

(get—baseO返回值)指明在CPU线性地址空间中起始基地址，第2个（get—limit()返回值）

说明欲释放的字节长度值。get—base()宏中的current-}ldt[l]给出进程代码段描述符的位置

(current->ldt[2]给出进程代码段描述符的位置)；get—limit()中的0x0f是进程代码段的

选择符（0x17是进程数据段的选择符）。即在取段基地址时使用该段的描述符所处地址作为

参数，取段长度时使用该段的选择符作为参数。free—page—tables()函数位于mm/memory.c

文件的 105 行，get—base()和 get—limit()宏位于 include/linux/sched.h 头文件的 213 行处。

\*/

free\_page\_tables (get\_base (current->ldt[1]), get\_limit (0x0f));

free\_page\_tables (get\_base (current->ldt[2]), get\_limit (0x17));

/\* 如果当前进程有子进程，就将子进程的father 置为1(其父进程改为进程1)。如果该子进程已经 \*/

/\* 处于僵死(ZOMBIE)状态，则向进程1 发送子进程终止信号SIGCHLD。 \*/

for (i = 0; i < NR\_TASKS; i++)

if (task[i] && task[i]->father == current->pid)

{

task[i]->father = 1;

if (task[i]->state == TASK\_ZOMBIE)

/\* assumption task[1] is always init \*/ /\* 这里假设 task[l]肯定是进程 init \*/

(void) send\_sig (SIGCHLD, task[1], 1);

}

/\* 关闭当前进程打开着的所有文件。 \*/

for (i = 0; i < NR\_OPEN; i++)

if (current->filp[i])

sys\_close (i);

/\* 对当前进程工作目录pwd、根目录root 以及运行程序的i 节点进行同步操作，

放回各个i节点并分别置空（释放）。

\*/

iput (current->pwd);

current->pwd = NULL;

iput (current->root);

current->root = NULL;

iput (current->executable);

current->executable = NULL;

/\* 如果当前进程是领头(leader)进程并且其有控制的终端，则释放该终端。 \*/

if (current->leader && current->tty >= 0)

tty\_table[current->tty].pgrp = 0;

/\* 如果当前进程上次使用过协处理器，则将last\_task\_used\_math 置空。 \*/

if (last\_task\_used\_math == current)

last\_task\_used\_math = NULL;

/\* 如果当前进程是leader 进程，则终止所有相关进程。 \*/

if (current->leader)

kill\_session ();

/\* 把当前进程置为僵死状态，表明当前进程已经释放了资源。并保存将由父进程读取的退出码。 \*/

current->state = TASK\_ZOMBIE;

current->exit\_code = code;

/\* 通知父进程，也即向父进程发送信号SIGCHLD -- 子进程将停止或终止。 \*/

tell\_father (current->father);

schedule (); /\* 重新调度进程运行，以让父进程处理僵死进程其他的善后事宜。

下面return语句仅用于去掉警告信息。因为这个函数不返回，所以若在函数名前加关键字

volatile,就可以告诉gcc编译器本函数不会返回的特殊情况。这样可让gcc产生更好一

些的代码，并且可以不用再写这条return语句也不会产生假警告信息。

\*/

return (-1); /\* just to suppress warnings \*/ /\* 仅用于去掉警告信息 \*/

}

/\* 系统调用exit()。终止进程。

参数error—code是用户程序提供的退出状态信息，只有低字节有效。把error—code左移8

比特是wait()或waitpidO函数的要求。低字节中将用来保存wait()的状态信息。例如，

如果进程处于暂停状态（TASK—STOPPED），那么其低字节就等于0x7f。参见sys/wait.h

文件第13—18行。wait()或waitpidO利用这些宏就可以取得子进程的退出状态码或子

进程终止的原因（信号）。

\*/

int

sys\_exit (int error\_code)

{

return do\_exit ((error\_code & 0xff) << 8);

}