# 进程间通讯的主要方式：

# 管道方式：

# 信号：

信号在最早的Unix系统中即被引入，用于在用户态进程间通信。内核也用信号通知进程系统所发生的事件。

## 信号的作用：

1. 信号是很短的消息，可以被发送到一个进程或一组进程。发送给进程的唯一信号通常是一个数。以此来标识信号。在标准信号中，对参数、消息或者其他相随的信息没有给予关注。
2. 名字前缀为SIG的一组宏用来标识信号。在Linux中，SIGCHLD宏的扩展为值17，当某一个子进程停止或终止时，SIGCHLD宏发送给父进程的信号标识符。SIGSEGV宏扩展为值11，当一个进程引用无效的内存时，SIGSEGV宏产生发送给进程的信号标识符。
3. 使用信号的两个主要目的是：
4. 让进程知道已经发生了一个特定的事件
5. 强迫进程执行它自己代码中的信号处理程序
6. Linux/I386中的前31个信号 P434，除了在这张表中描述的常规信号，POSIX标准还引入了一类新的信号，叫做实时信号，在Linux中它们的编码范围为32~64。它们与常规信号有很大的不同，因为它们必须排序以便发送的多个信号能被接收到。另一方面，同种的常规信号并不排队：如果一个常规信号被连续发送多次，那么，只有其中的一个发送到接收进程。尽管Linux内核并使用实时信号，它还是通过几个特定的系统调用完全实现了POSIX标准。
7. 许多系统调用允许程序员发送信号，并决定他们的进程如何响应所接收的信号。P435 表11-2
8. 信号的一个重要特点是它们可以随时被发送给状态经常不可预知的进程。发送非运行的进程的信号必须由内核保存，直到进程恢复执行。阻塞一个信号要求信号的传递拖延，直到随后解除阻塞，这使得信号产生一段时间之后才能对其传递。
9. 内核区分信号传递的两个不同阶段：
10. 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被发送
11. 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，或两个都做。
12. 每个所产生的信号至多被传递一次。信号是可消费资源，一旦被消费了，或者说一旦它们被传递出去了，进程描述符中有关这个信号的所有信息都被取消。下次自然也就不能再使用。
13. 已产生但还没有传递的信号称为挂起信号。任何时候，一个进程仅存在给定类型的一个挂起信号，同一种类型的其他信号不被排队，只被简单地丢弃。但是，实时信号是不同的，同种类型的挂起信号可以有好几个。

## 传递信号前所执行的操作

1. 进程以三种方式的其中一个对一个信号做出应答：
2. 显示地忽略信号
3. 执行与信号相关的缺省操作。由内核预定义的缺省操作取决于信号的类型，可以是下列类型之一：

Terminate：进程被终止（杀死）

Dump：进程被终止（杀死），并且，如果可能，创建包含进程执行上下文的核心转存文件；这个文件可以用于调试。

Ignore：信号被忽略

Stop：进程被停止，即把进程置为TASK\_STOPPED状态

Continue：如果进程被停止（TASK\_STOPPED），就把它置为TASK\_RUNNING状

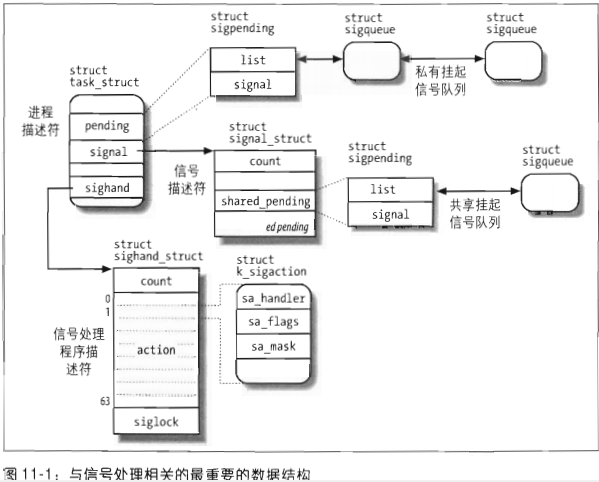
1. 通过调用相应的信号处理程序函数捕获信号。
2. 一个信号被阻塞和忽略是不同的：只要信号被阻塞，他就不被传递；只有在信号解除阻塞后才能传递它。而一个信号被忽略，依然能被传递，只是没有进一步的操作。
3. SIGKILL和SIGSTOP信号不可以被显示地忽略、捕获或阻塞，因此，通常必须执行它们的缺省操作（目标进程无论如何都会收到这种信号）。因此，SIGKILL和SIGTOP运行具有适当特权的用户分别终止并停止任何进程，不管程序采取怎样的防御措施。
4. 如果信号的传递会引起内核杀死一个进程，那么这个信号对该进程就是致命的。SIGKILL信号总是致命的；而且，缺省操作为Terminate的每个信号，以及不被进程捕获的信号对该进程也是致命的。注意，如果一个被进程所捕获的信号，其对应的信号处理函数终止了这个进程，那么这个信号就不是致命的，因为进程自己选择了终止，而不是被内核杀死。

## POSIX信号和多线程应用

1. 如果一个挂起信号发送给了某个特定进程，那么这个信号是私有的；如果发送给了整个线程组，它就是共享的。 P438

## 与信号相关的数据结构

1、信号相关的数据结构：P439



2、与信号处理相关的进程描述符中的字段：P440

类型 名称 说明

struct signal\_struct\* signal 指向进程所拥有的信号描述符的指针

struct sighand\_struct\* sighand 指向进程所拥有的信号处理程序描述符的指针

sigset\_t[] blocked 被阻塞信号的掩码

sigset\_t real\_blocked 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的地址

size\_t sas\_ss\_size 信号处理程序备用堆栈的大小

int (\*)(void\*) notifier 指向一个函数的指针，设备驱动程序用这个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函数所阻塞的信号的位掩码

1. blocked字段存放当前进程所屏蔽的信号。它是一个sigset\_t位数组，每种信号类型一个元素：typedef struct( unsigned long sig[2] ) sigset\_t;

## 信号描述符和信号处理程序描述符

1、信号描述符中和信号处理有关的字段： P441

类型 名称 说明

atomic\_t count 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

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

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 存放共享挂起信号的数据结构

int group\_exit\_code 线程组的终止代码

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

int notify\_count 在杀死整个线程组的时候使用

int group\_stop\_count 在杀死整个线程组的时候使用

unsigned int flags 在传递修改进程状态的信号时使用的标志

2、信号处理程序描述符： P441

类型 名称 说明

atomic\_t count 计数器，共享该结构的进程个数

strcut k\_sigaction[64] action 说明在所传递信号上执行操作的数组

spinlock\_t siglock 保护信号描述符和信号处理程序描述符的自旋锁

1. 在调用clone()系统调用时设置CLONE\_SIGHAND标志，信号处理程序描述符就可以由几个进程共享。
2. 在POSIX的多线程应用中，线程组中的所有轻量级进程都引用相同的信号描述符和信号处理程序描述符。

## sigaction数据结构

1. k\_sigaction结构既包含对用户态进程所隐藏的特性，也包含sigaction结构，该结构保存了用户态进程能看见的所有特性。在80x86平台上，信号的所有特性对用户态的进程都是可见的。因此，k\_sigaction结构只不过简化为类型为sigaction的结构，该结构包含下列字段：
2. sa\_handler：这个指定要操作执行操作的类型。它的值可以是指向信号处理程序的一个指针，SIG\_DFL（即值0，指定执行缺省操作），或者SIG\_IGN（即值1，指定忽略信号）。
3. sa\_flags：这是一个标志集，指定必须怎样处理信号。其中的一些标志在表11-6列出
4. sa\_mask：这是类型为sigset\_t的变量，指定当运行信号处理程序时要屏蔽的信号
5. 表11-6：指定如果处理信号的一组标志 P442

## 挂起信号队列

1. 为了跟踪当前的挂起信号是什么，内核把两个挂起信号队列与每个进程相关联：
2. 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。
3. 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. 挂起信号队列由sigpending数据结构组成，它的定义如下：

struct sigpending{ struct list\_head list;

sigset\_t signal };

// signal字段是指定挂起信号的位掩码，而list字段包含sigqueue数据结构的双向链表的头

1. sigqueue数据结构的字段 P443

struct list\_head list 链接挂起队列的链表

spinlock\_t\* lock 指向与挂起信号对应的信号处理程序描述符中siglock字段的指针

Int flags Sigqueue数据结构的标志

siginfo\_t info 描述产生信号的事件

struct user\_struct\* user 指向进程拥有者的每用户数据结构的指针

4、siginfo\_t是一个128字节的数据结构，其中存放有关出现特定信号的信息。P444

5、表11-8：最重要的信号发送者代码 P444

## 在信号数据结构上的操作

1. sigemptyset(set)和sigfillset(set)：把sigset\_t类型的变量中的位分别置为0或1
2. sigaddset(set, nsig)和sigdelset(set, nsig)：把nsig信号在sigset\_t类型变量中对应的位分别置为1或0.
3. sigaddsetmask(set, mask)和sigdelsetmask(set, mask)把mask中的位在sigset\_t类型变量中对应的所有位分别设置为1或0。它们仅用于编号为1~32之间的信号。
4. 剩余P445

## 产生信号

1. 当发送给进程一个信号时，这个信号可能来自内核，也可能来自另外一个进程。内核通过对如表11-9所示的某个函数进行调用而产生信号： P446
2. 当发送给线程组一个信号时，这个信号可能来自内核，也可能来自另外一个进程。内核对如表11-10所示的某个函数调用产生这类信号。P447
3. specific\_send\_sig\_info()函数向指定进程发送信号 P447
4. send\_signal()函数：在挂起信号队列中插入一个新元素 P449
5. group\_send\_sig\_info()函数：向整个线程组发送信号 P450

## 传递信号

1. do\_signal()函数：处理非阻塞的挂起信号 P453

## 执行信号的缺省操作

1. 详细从**P454**开始
2. SIGSTOP与其他信号的差异比较微妙：SIGSTOP总是停止线程组，而其他信号只停止不在“孤儿进程组”中的线程组。POSIX标准规定，只要进程组中有一个进程有父进程，尽管父进程处于不同的进程组中但和子进程在同一个会话中，那么这个进程就不是孤儿。因此，如果父进程死亡，但启动该进程的用户仍然登录在线，那么该进程组就不是一个孤儿。
3. 缺省操作为Dump的信号可以在进程的工作目录中创建一个“转存”文化，这个文件列出进程地址空间和CPU寄存器的全部内容。do\_signal()创建转储文件后，就杀死这个线程组。剩余18个信号的缺省操作是Terminate，它仅仅是杀死线程组。

## 捕获信号

1、如果信号有一个专门的处理程序，do\_signal()函数就必须强迫该处理程序执行。P455

# 消息队列：

# 共享内存：

# 套接字：

# 相关概念：

1. 进程
2. 概念：程序执行时的一个实例 或 一个运行程序的”执行上下文“。
3. 在Linux源码中，常把进程称为任务（task）或线程（thread）。从内核观点来看，进程的目的就是担当分配系统资源（CPU时间、内存等）的实体。
4. 当一个进程创建时，它几乎与父进程相同。它接受父进程地址空间的一个（逻辑）拷贝，并从进程创建系统调用的下一条指令开始执行与父进程相同的代码。尽管父子进程可以共享程序代码的页，但是它们各自有独立的数据拷贝（栈和堆），因此，子进程对一个内存单元的修改对父进程是不可见的（反之亦然）。
5. 地址空间：运行进程引用的内存地址集合
6. 多道程序系统（多处理系统）：允许进行并发活动，并能竞争系统资源的系统。
7. 区分程序和进程：几个进程能并发的执行一个程序，而同一个进程能顺序地执行几个进程。
8. 调度程序：决定哪个进程能执行。
9. 硬链接（链接）：包含在目录中的文件名。在同一个目录或者不同的目录中，同一个文件可以有几个链接，因此对应几个文件名
10. 多线程应用程序：
11. 拥有很多相对独立执行流的用户程序共享应用程序的大部分数据结构。一个进程由几个用户线程（或简单地说，线程）组成，每个线程都代表进程的一个执行流。现在，大部分多线程应用程序都是用pthread(POSIX thread)库的标准库函数集编写的。
12. Linux使用轻量级进程（lightwetght process）对多线程应用程序提供更好的支持。两个轻量级进程基本上可以共享一些资源，诸如地址空间、打开的文件等等。只要其中一个修改共享资源，另外一个就立即查看这种修改。当然，当两个线程访问共享资源时，就必须同步它们自己。
13. 实现多线程应用程序的一个简单方式就是把轻量级进程与每个线程关联起来。这样，线程之间就可以通过简单地共享同一内存地址空间、同一打开文件集等来访问相同的应用程序数据结构集。同时，每个线程都可以由内核独立调度，以便一个睡眠的同时，另外一个仍然可以运行。

## 进程描述符：

1. 为了管理进程，内核必须对每个进程所做的事情进行清楚的描述。进程描述符都是task\_struct类型结构，它的字段包含了与一个进程相关的所有信息。进程描述符不仅包含了进程属性的字段，而且一些字段还包含了指向其他数据结构的指针。
2. 一般来说，能被独立调度的每个执行上下文都必须拥有它自己的进程描述符。因此，因此，即使共享内核大部分数据结构的轻量级进程，也都有它们自己的task\_struct结构。32位进程描述符地址来标识进程。进程描述符指针指向这些地址，内核对进程的大部分引用是通过进程描述符指针进行的。
3. 另一方面，Unix操作系统允许用户使用一个叫做进程标识符processID（或PID）的数来标识进程，PID存放在进程描述符的pid字段中。PID被循序编号，新创建的进程的PID通常是前一个进程的PID加1。不过PID的值有一个上限，当内核使用的PID达到这个上限值的时候，就必须开始循环使用已经闲置的小PID号。在默认情况下，最大的PID号32767（PID\_MAX\_DEFAULT - 1）；
4. 由于循环使用PID编号，内核必须通过管理一个pidmap\_array位图来标识当前已经分配的PID和闲置的PID号。Linux把不同的PID和系统中每个进程或轻量级进程相关联。这中方式能提供最大的灵活行，因为系统中每个执行上下文都可以被唯一的识别。
5. Linux引入线程组的表示，一个线程组中的所有线程和使用该线程组的领头线程相同的PID，也就是该组中第一个轻量级进程的PID，该组中第一个轻量级进程的PID被存入进程描述符的tgid字段中。getpid（）系统调用返回当前进程的tgid值而不是pid的值。因此一个多线程应用的所有线程共享相同的PID。绝大多数进程属于一个线程组，线程组包含单一的成员；线程组的领头线程其tgid的值与pid的值相同，因而getpid（）系统调用对这类进程所起的作用和一般进程是一样的。
6. 进程描述符通过task和thread\_info字段和线程描述符相关联，内核使用alloc\_thread\_info和free\_thread\_info宏分配和释放 存储thread\_info结构和内核栈的内存区。
7. 为了获取当前在CPU上运行进程的进程描述符的指针，内核要调用current宏。
8. 用栈存放进程描述符。对于每个CPU，仅通过检查栈就可以获得当前正确的进程。
9. 字段：

类型 名称 说明

unsigned long thread\_info->flags 存放TIF\_NEED\_RESCHED标志，如果必须调用调度程序，则设置该标志

unsigned int thread\_info->cpu 可运行进程所在运行队列的CPU逻辑号

unsigned long state 进程的当前状态

int prio 进程的动态优先级

int static\_prio 进程的静态优先级

struct list\_head run\_list 指向进程所属的运行队列链表中的下一个和前一个元素

prio\_array\_t\* array 指向包含进程的运行队列

unsigned long sleep\_avg 进程的平均睡眠时间

unsigned long long timestamp 进程最近一次插入运行队列的时间，或涉及本进程最后一次进程切换的时间（包含进程进入睡眠状态的进程切换的时间撮）

unsigned long long last\_can 最近一次提哈un本进程的进程切换时间

int active 进程被唤醒时所用的条件代码

unsigned long policy 进程的调度类型(SCHED\_NORMAL、SCHED\_RR或SCJED\_FIFO)

cpumask\_t cpus\_allowed 能执行进程的CPU的位掩码

unsigned int time\_slice 在进程的时间片中还剩余的时钟节拍数

unsigned int first\_time\_slice 如果进程肯定不会用完其时间片，就把该标志设置为1

unsigned long rt\_priority 进程的实时优先级

1. 进程状态：进程描述符中的state字段描述了进程当前所处的状态。它由一组标志组成，其中每个标志描述一种可能的进程状态。

下面是进程可能的状态：

1. 可运行状态（TASK\_RUNNING）：进程要么在CPU上执行，要么准备执行。只有在该状态的进程才可能在CPU上运行。
2. 可中断的等待状态（TASK\_INTERRUPTIBLE）：进程被挂起（睡眠）。直到某个条件变为真。产生一个硬件中断，释放进程正在等待的系统资源，或传递一个信号都是可以唤醒进程的条件（把进程的状态放回到TASK\_RUNNING）。

处于这个状态的进程因为等在某些事件的发生（比如等待socket连接、等待信号量），而被挂起。这些进程的task\_struct结构被放入对应事件的等待队列中。当这些事件发生时（由外部中断触发、或由其他进程触发），对应的等待队列中的一个或多个进程将被唤醒。

1. 不可中断的等待状态（TASK\_UNINTERRUPTIBLE）：与可中断的等待状态相似，把信号传递到睡眠进程不能改变它的状态。这种状态很少用到，但在一些特定的条件下，这种状态是很有用的。比如，当进程打开一个设备文件，其相应的设备驱动程序开始探测相应的硬件设备时会用到这种状态。探测完成以前，设备驱动程序不能被中断，否则，硬件会处于不可预知的状态。
2. 暂停状态（TASK\_STOPPED）：进程执行被暂停。当进程接收到SIGSTOP、SIGTITP、SIGTTIN或者SIGTTOU信号后，进入暂停状态。向进程发送一个SIGCONT信号，可以让其从暂停状态恢复到可执行状态。
3. 跟踪状态（TASK\_TRACED）：进程的执行已由debugger程序暂停。当一个进程监控时（例如dubugger执行ptrace（）系统调用监控一个测试程序），任何信号都可以把这个进程置于TASK\_TRACED状态。当进程正在被跟踪时，它处于跟踪状态。

对于进程本身来说，暂停状态和跟踪状态很类似，都是表示进程暂停下来。而跟踪状态相当于在暂停状态上多了一层保护，处于跟踪状态的进程不能响应SIGCONT信号而被唤醒。只能等到调试进程ptrace系统调用执行ptrace\_cont、ptrace\_detach等操作，或调试进程退出，被调试进程才能恢复可执行状态。

1. 僵死状态（EXIT\_ZOMBIE）：进程的执行已经被终止了，但是，父进程还没有发布wait4（）或waitpid（）系统调用来返回有关死亡进程的消息。发布wait（）类系统调用前，内核不能丢弃包含在僵死进程描述符中的数据，因为父进程可能还需要它。
2. 僵死撤销状态（EXIT\_DEAD）：为了防止其他执行线程在同一进程上也执行wait（）类系统调用，而把进程的状态由僵死状态改为僵死撤销状态。

10、list\_head：linux使用list\_head来生成双向链表。

11、进程链表：

（1）进程链表把所有进程的描述符连接起来。每个进程描述符的task\_struct结构都包含一个list\_head类型的tasks字段，这个类型的prev和next字段分别指向前面和后面的task\_stuct元素。

（2）进程链表的头是init\_task描述符，它是所谓的0进程或者swapper进程的进程描述符。Init\_task的task.prev字段指向链表中最后插入的进程描述符的tasks字段。

（3）SET\_LINKS和REMOVE\_LINKS宏分别用于从进程中链表中插入和删除一个进程描述符。

12、TASK\_RUNNING状态的进程链表：

13、进程上下文：当一个进程执行时，CPU的所有寄存器的值、进程的状态以及堆栈中的内容被称为进程上下文。

14、SMP：多处理系统（系统中含有多个CPU）

15、自旋锁：自旋锁是一个互斥设备，它只有两个值：“锁定”和“解锁”。如果锁可用，则“锁定”，而代码继续进入临界区；相反，如果锁被其他人获得，则代码进入忙循环，并重复检查这个锁，直到该锁可以用为止。这就是自旋过程。

## 进程间关系：

1. 进程和它所创建的子进程具有父子关系。如果一个进程创建多个子进程，则子进程之间具有兄弟关系。
2. pidhash表：
3. 内核为了快速从进程的PID导出对应的进程描述符指针而采用的数据结构。
4. 顺序扫描进程链表并检查进程描述符的pid字段是可行但相当低效的，为了加速查，引入了4个散列表。需要4个散列表是因为进程描述符包含了表示不同类型PID的字段，而且每种类型的PID需要它自己的散列表。
5. 内核初始化期间动态地为4个散列表分配空间，并把它们的地址存入pid\_hash数组。
6. Pid\_hashfn宏把PID转化为表索引。但是不同的PID转化成表索引可能相同，导致冲突。Linux利用链表来处理冲突的PID：每一个PID\_hash表项是由冲突的进程描述符组成的双向链表。为了能快速返回组中其他所有进程，内核为每个线程组保留一个进程链表。

## 等待队列：

1. 等待队列实现了在事件上的条件等待：希望等待特定事件的进程把自己放入适合的等待队列，并放弃控制器权。因此，等待队列表示一组睡眠的进程，当某一条件变为真时，由内核唤醒它们。
2. 等待队列由双向链表实现，其元素包括指向进程描述符的指针。
3. 等待队列是由终端处理程序和主要内核函数修改的，因此必须对其双向链表进行保护以免对其进行同时访问，因为同时访问会导致不可预测的后果。同步是通过等待队列中的lock自旋锁达到的。Task\_list字段是等待进程链表的头。
4. 等待队列的元素类型：P114
5. 非互斥进程总是在等待队列的开始位置，互斥进程的总是在双向链表额尾部。
6. void interruptible\_sleep\_on(struct wait\_queue\_head\_t wq);把当前进程放入参数所指向的等待队列中。函数最后会调用schedule（），当前进程被挂起，让CPU运行别的进程。
7. wake\_upinterruptible(struct wait\_queue\_head\_t wq); 把在等待队列中的进程唤醒。
8. 利用上面两个函数可以实现生产者消费者模式。

## 进程资源限制：

1. 每个进程都有一组相关的资源限制，限制指定了系统资源数量。这些限制能避免用户过分使用系统资源（CPU、磁盘空间等）。

## 进程切换：

1. 概念：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，这种行为叫做进程切换、任务切换或上下文切换。
2. 尽管每个进程可以拥有自己的地址空间，但所有的进程必须共享CPU寄存器。比如在挂起进程时，会将CPU寄存器中的值保存起来，然后在恢复这个进程的执行之前，内核必须确保每个寄存器装入了挂起进程时的值。
3. 硬件上下文：进程恢复执行前必须装入寄存器的一组数据称为硬件上下文。硬件上下文是进程可执行上下文的一个子集，因为可执行上下文包含进程执行时需要的所有信息。在Linux中，进程硬件上下文的一部分存放在TSS段（任务状态描述段），而剩余部分存放在内核态堆栈中。
4. 进程切换只发生在内核态。在执行进程切换之前，用户态进程使用的所有寄存器内容都已经保存在内核态堆栈中，这也包括ss和esp这对寄存器的内容（存储用户态堆栈指针的地址）。
5. 任务状态段TSS：
6. 用来存放硬件上下文。Linux强制为系统中每个不同的CPU创建一个TSS。Init\_tss数组为系统上每个不同的CPU存放一个TSS。
7. 每个TSS都有它自己8字节的任务状态段描述符（TSSD）。这个描述符包括指向TSS起始地址的32位Base字段，20位Limit字段。TSSD的S标志位被清0，以表示相应的TSS是系统段的事实。
8. 由Linux创建的TSSD存放在全局描述符表（GDT）中，GDT的基地址存放在每个CPU的gdtr寄存器中。每个CPU的tr寄存器包含相应TSS的TSSD选择符，也就包含TSSD的Base字段和Limit字段。这样，处理器就能直接对TSS寻址而不用从GDT中检索TSS的地址。
9. 进程描述符的thread字段：每次进程切换时，被替换进程的硬件上下文必须保存在别处。只要进程被切换出去，内核就把其硬件上下文保存在这个结构中。
10. 进程切换的步骤：
11. 切换页全局目录以安装一个新的地址空间
12. 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要的所有信息，包括CPU寄存器。

// 进程切换的函数 switch\_to() 和 \_\_switch\_to() P125页

1. 日后重看P128 保存和加载FPU、MMX及XMM寄存器

## 进程的创建：

1. 在Linux中，轻量级进程是由clone的函数创建的，这个函数使用下列参数：
2. 实际上，clone（）是C语言库中定义的一个封装函数，它负责建立新轻量级进程的堆栈并且调用对编者隐藏的clone（）系统调用。实现clone（）系统调用的syn\_clone()服务例程没有fn和arg参数。实际上，封装函数把fn指针存放在子进程堆栈的某个位置处，该位置就是该封装函数本身返回地址存放的位置。Arg指针正好存放在子进程堆栈中fn的下面。当封装函数结束时，CPU从堆栈中取出返回地址，然后执行fn（arg）函数。
3. 传统的fork（）系统调用在Linux中使用clone（）实现的，其中clone（）的flags参数指定为SIGCHLD信号及所有清0的clone标志，而它的child\_stack参数是父进程当前的堆栈指针。 因此，父进程和子进程暂时共享同一个用户堆栈。但是，要感谢写时复制机制，通常只要父子进程中有一个试图去改变栈，则立即各自得到用户态堆栈的一份拷贝。
4. 写时复制技术允许父子进程读相同的物理页。只要两者中有一个试图写一个物理页，内核就把这个页的内容拷贝到一个新的物理页，并把这个新的物理页分配到正在写的进程。
5. vfork()系统调用创建的进程能共享父进程的内存地址空间。为了防止父进程重写子进程需要的数据，阻塞父进程的执行，一直到子进程退出或执行一个新的程序为止。vfork()系统调用在Linux中也是clone()实现的，其中clone()的参数flags指定为SIGCHLD信号和CLONE\_VM及CLONE\_VFORK标志，clone()的参数child\_stack等于父进程当前的栈指针。
6. do\_fork()利用辅助函数copy\_process()来创建进程描述符以及子进程执行所需要所有其他内核数据结构。
7. copy\_process()创建进程描述符以及子进程执行所需要所有其他内核数据结构。

## 内核线程：

1. 内核线程不同于普通线程的地方：
2. 内核线程只运行在内核态，而普通线程可以运行在内核态，也可以运行在用户态。
3. 因为内核线程只运行在内核态，它们只使用大于PAGE\_OFFSET的线性地址空间。另一方面，不管在用户态还是内核态，普通进程可以用4GB的线性地址空间。
4. kernel\_thread()函数用于创建一个新的内核线程。

## 进程0：

1. 概念：所有进程的祖先叫做进程0，idle进程 或 因为历史原因叫做swapper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。它使用静态分配的数据结构，所有其他进程的数据结构都是动态分配的。
2. 在多处理器系统中，每个CPU都有一个进程0。只要打开机器电源，计算机的BIOS就启动某一个CPU，同时禁用其他CPU。运行在CPU 0上swapper进程初始化内核数据结构，然后激活其他的CPU，并通过copy\_process()函数创建另外的swapper进程，把0传递给新创建的swapper进程作为它们的新PID。此外，内核把适当的CPU索引赋值给内核所创建的每个进程的thread\_info描述符的cpu字段。

## 进程1：

1. 概念：由进程0创建的内核进程，PID为1，并与进程0共享所有的内核数据结构。当调度程序选择到它时，init进程开始执行init（）函数。依次完成内核初始化。Init()调用execve（）系统调用装入可执行程序init。结果，init内核线程变成一个普通进程，并拥有自己的每进程（per-process）内核数据结构。在系统关闭前，init进程一直存活，因为它创建和监控在操作系统外层执行的所有进程的活动。

## 撤销进程：

1. 进程终止的一般方式是调用exit()库函数，该函数释放C函数库所分配的资源。C编译程序把exit（）函数插入到main（）函数的最后一条语句之后。
2. 内核可以有选择地强迫整个线程组死掉。这发生一下两种典型情况下：
3. 当进程接收到一个不能处理或忽略的信号时
4. 当内核正在代表进程运行时，在内核态产生了一个不可恢复的CPU异常。
5. do\_group\_exit函数杀死属于current线程组的所有进程。
6. do\_exit()函数：所有进程的终止都是由do\_exit()函数来处理的。这个函数从内核数据结构中删除对终止进程的大部分引用。

## 进程删除：

1. 引入僵死进程的原因：从技术上进程已死，但必须保留它的描述符，直到父进程得到通知。
2. 如果父进程在子进程结束之前结束，系统中会到处是僵死进程，而且它们的进程描述符会永远占据RAM。因此，必须强迫所有的僵死进程成为init进程的子进程来解决这个问题。这样，init进程在用wait（）类系统调用检查其合法的子进程终止时，就会撤销僵死的进程。

## 守护进程：

1、概念：守护进程是一种运行在后台的特殊进程，它独立于控制终端并且周期性的执行某种任务或者等待处理某些发生的事件。由于在Linux中，每个系统与用户交互都依赖于这个终端，这个终端被称为这些进程的控制终端，当控制终端被关闭的时候，相应的进程都会自动关闭。但是守护进程能够突破这种限制，它脱离终端并且在后台运行，并且它脱离的目的是为了避免进程在运行的过程中信息在任何终端中显示，并且进程也不会被任何终端所产生的信息所打断。它从被执行的时候开始运转，直到整个系统关闭才退出（当然可以人为的杀死相应的守护进程）。如果想让某个进程不因为用户、中断、或者其他变化而影响，那么就必须把这个进程变成守护进程。

2、步骤：http://blog.csdn.net/liangxanhai/article/details/7752898

（1）创建子进程，父进程退出（使子进程变成孤儿进程）：父进程是从shell终端开启的，当父进程退出以后，会在shell终端里造成了一个程序已经运行完毕的假象。之后的所有工作在子线程中完成，而用户在shell终端里可以执行其他命令，从而在形式上就做到了和控制终端脱离。

（2）在子进程中创建新的会话（脱离控制终端）：

由于在父进程中调用了fork()函数，子进程拷贝了父进程的会话期、进程组、控制终端等资源，虽然父进程退出了，但是会话期、进程组、控制终端等并没有改变，因此，需要使用setsid()函数来使该子进程完全独立出来，从而摆脱其他进程的控制。

Setsid()函数可以：

1. 创建一个新的会话，并且该进程担任会话组的组长。
2. 改变当前目录为根目录
3. 重设文件的权限掩码
4. 关闭文件描述符

## 相关函数：

1、signal\_pending() ：检查当前进程是否有信号需要处理