# 数据结构

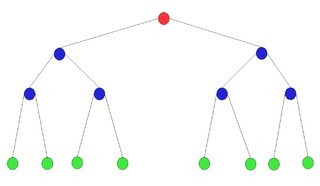
## 链表

## 二叉树

### 二叉树的定义

二叉树是每个节点最多有两个子树的树结构。通常子树被称作“左子树”（left subtree）和“右子树”（right subtree）。二叉树常被用于实现二叉查找树和二叉堆。

二叉树的每个结点至多只有二棵子树(不存在度大于2的结点)，二叉树的子树有左右之分，次序不能颠倒。



### 特殊二叉树

1. 斜树

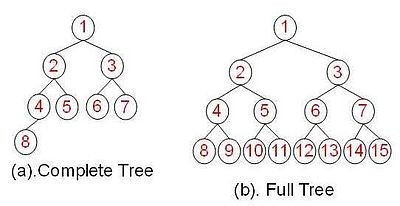
所有结点都只有左子树的二叉树叫左斜树，所有结点都只有右子树的二叉树叫右斜树。斜树的每一层都只有一个结点，结点的个数与斜树的深度相同。

2. 满二叉树

在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子结点都在同一层上，这样的二叉树称为满二叉树。（上图中所示的二叉树，就是一棵满二叉树）

3. 完全二叉树

对一棵具有n个结点的二叉树按层序编号，如果编号为i（1≤i≤n）的结点与同样深度的满二叉树中的编号为i的结点在二叉树中的位置完全相同，则这棵二叉树称为完全二叉树。



**二叉搜索树(BST)**

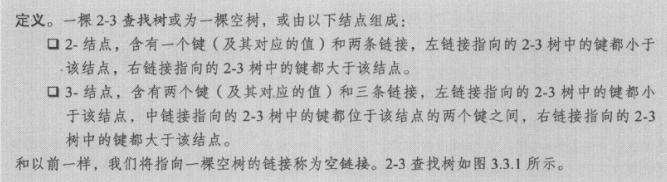
二叉搜索树的特点是左子树的值<=父节点的值<=右子树的值。

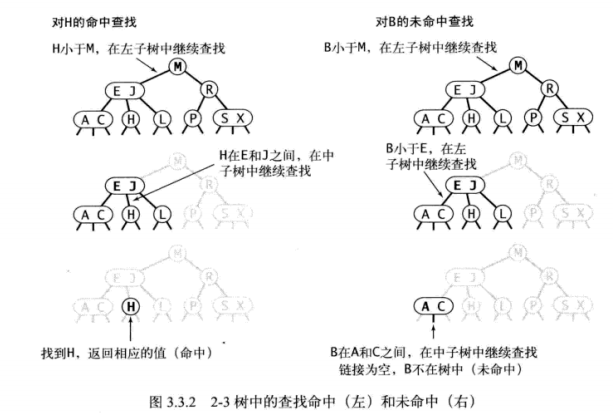
**平衡二叉树(BBT)**

平衡二叉树的特点是左右子树的深度差不能大于1

2-3二叉树

<http://blog.csdn.net/guanhang89/article/details/51039673>

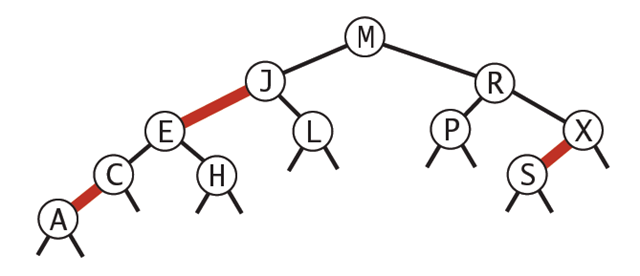


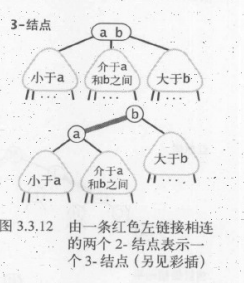


**红黑二叉树**

**See** [**http://www.cnblogs.com/yangecnu/p/Introduce-Red-Black-Tree.html**](http://www.cnblogs.com/yangecnu/p/Introduce-Red-Black-Tree.html)

基本思想是用标准的二叉查找树和一些额外的信息（替换3-结点）来表示2-3树。红链接将两个2-结点连起来构成一个3-结点，黑链接则是2-3树中的普通链接。确切地说，我们将3-结点表示为一条由左斜的红色链接。



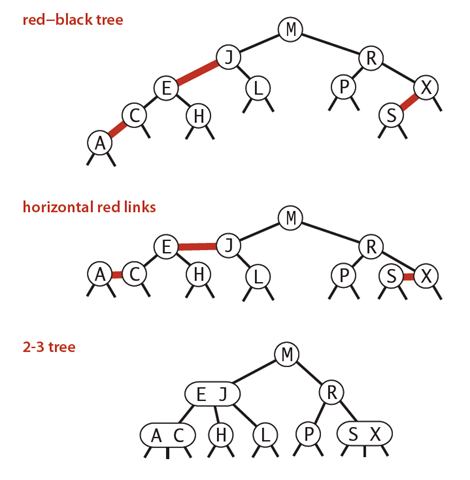


等价含义

红色均为左链接；

没有任何一个结点同时和两条红色链接相连；

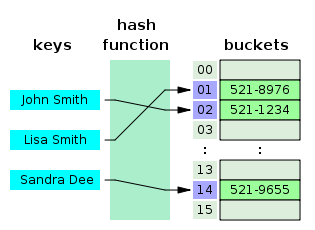
该树是完美黑色平衡的，即任意空链接到根节点的路径上的黑链接数量相同。



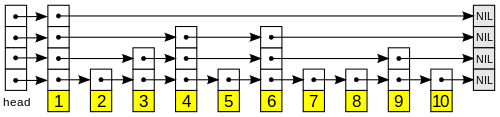
## 哈希表

[散列表](http://baike.baidu.com/view/1320746.htm)（Hash table，也叫哈希表），是根据关键码值(Key value)而直接进行访问的[数据结构](http://baike.baidu.com/view/9900.htm)。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做[散列函数](http://baike.baidu.com/view/131153.htm)，存放记录的[数组](http://baike.baidu.com/view/209670.htm)叫做[散列表](http://baike.baidu.com/view/1320746.htm)。

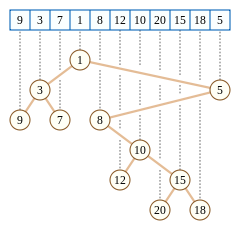
给定表M，存在函数f(key)，对任意给定的关键字值key，代入函数后若能得到包含该关键字的记录在表中的地址，则称表M为哈希(Hash）表，函数f(key)为哈希(Hash) 函数。



## Skip list



## Cartesian Tree



From <https://en.wikipedia.org/wiki/Cartesian_tree>

The Cartesian tree for a sequence of distinct numbers can be uniquely defined by the following properties:

1. The Cartesian tree for a sequence has one node for each number in the sequence. Each node is associated with a single sequence value.
2. A [symmetric (in-order) traversal](https://en.wikipedia.org/wiki/Tree_traversal) of the tree results in the original sequence. That is, the left subtree consists of the values earlier than the root in the sequence order, while the right subtree consists of the values later than the root, and a similar ordering constraint holds at each lower node of the tree.
3. The tree has the [heap property](https://en.wikipedia.org/wiki/Binary_heap): the parent of any non-root node has a smaller value than the node itself.[[1]](https://en.wikipedia.org/wiki/Cartesian_tree#cite_note-1)

## AVL tree

In [computer science](https://en.wikipedia.org/wiki/Computer_science), an **AVL tree** is a [self-balancing binary search tree](https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree)

The AVL tree is named after its two [Soviet](https://en.wikipedia.org/wiki/Soviet_Union) inventors, [Georgy Adelson-Velsky](https://en.wikipedia.org/wiki/Georgy_Adelson-Velsky) and [Evgenii Landis](https://en.wikipedia.org/wiki/Evgenii_Landis), who published it in their 1962 paper "An algorithm for the organization of information".[[3]](https://en.wikipedia.org/wiki/AVL_tree#cite_note-3)

## B-tree

The B-tree is a generalization of a [binary search tree](https://en.wikipedia.org/wiki/Binary_search_tree) in that a node can have more than two children ([Comer 1979](https://en.wikipedia.org/wiki/B-tree#CITEREFComer1979), p. 123). Unlike [self-balancing binary search trees](https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree), the B-tree is optimized for systems that read and write large blocks of data. B-trees are a good example of a data structure for external memory. It is commonly used in [databases](https://en.wikipedia.org/wiki/Database) and [filesystems](https://en.wikipedia.org/wiki/Filesystem).

A B-tree is kept balanced by requiring that all leaf nodes be at the same depth. This depth will increase slowly as elements are added to the tree, but an increase in the overall depth is infrequent, and results in all leaf nodes being one more node farther away from the root.

A **2-3 tree** is a B-tree.

## 排序算法

排序分类 平均时间 时间复杂度 辅助存储空间

简单排序 O(n2) O(n2) O(1)

快速排序 O(nlog2n) O(nlog2n) O(nlog2n)

堆排序 O(nlog2n) O(nlog2n) O(1)

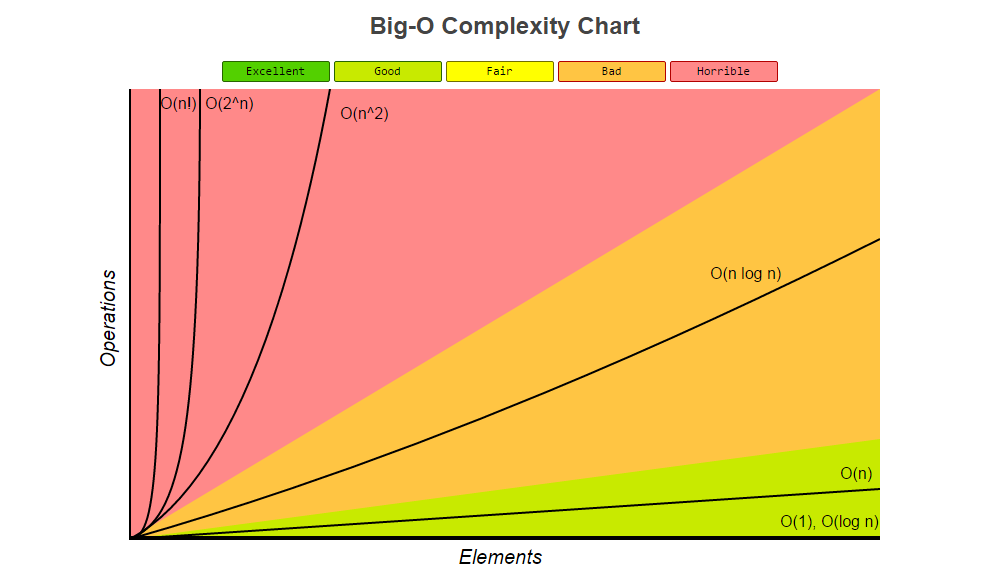
归并排序 O(nlog2n) O(nlog2n) O(n)

## 查找算法

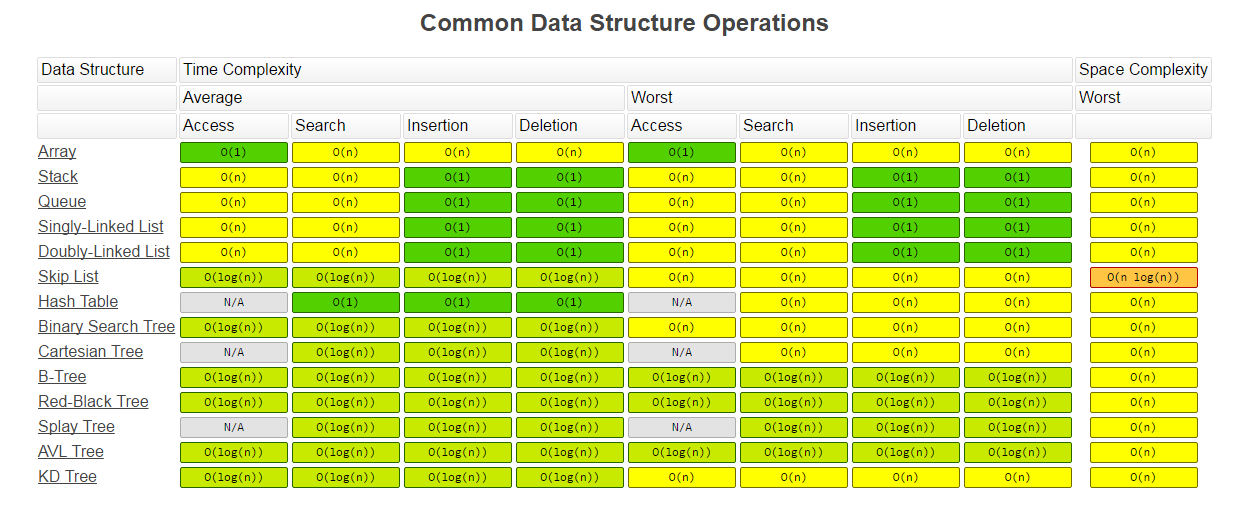
TBD

## 复杂度

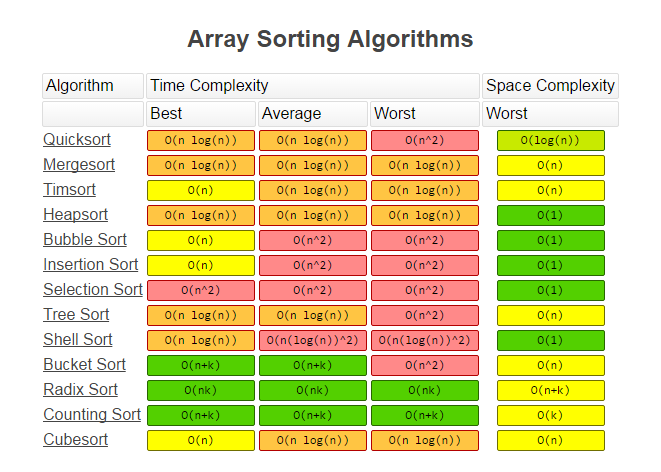
参考<http://bigocheatsheet.com/>



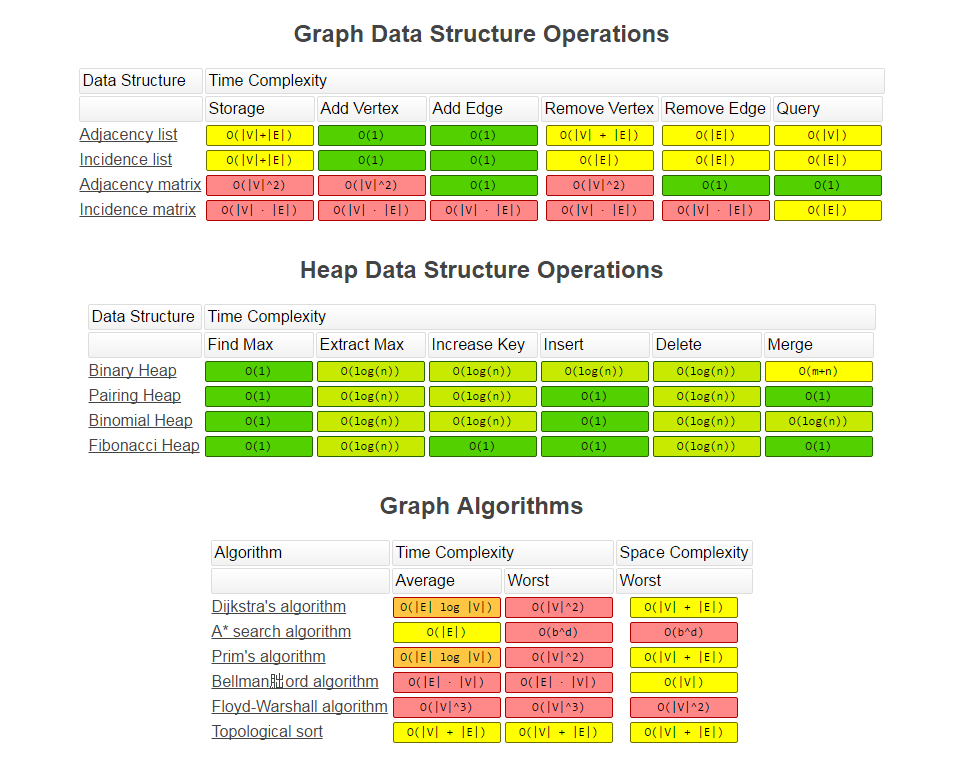
### Common Data Structure



### Array Sorting



### Others



# 操作系统

## Linux

### 系统头文件

**头文件目录：**

/usr/include/

--/sys

--/linux

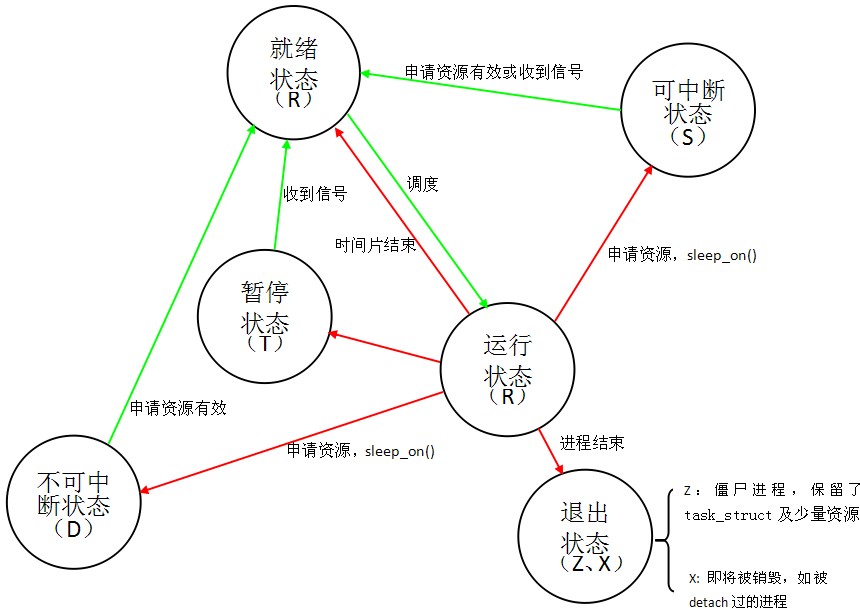
--/net

--…

**库文件：**

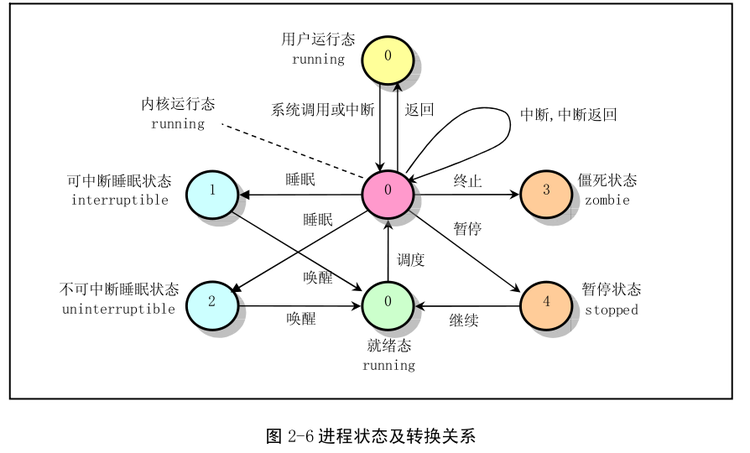
/usr/lib

### 进程的状态



RSTDZX

See <http://blog.csdn.net/u011068838/article/details/18216629>



**注意**

只有当进程从“内核运行态”转移到“睡眠状态”时，内核才会进行进程切换操作。在内核态下运行的进程不能被其它进程抢占，而且一个进程不能改变另一个进程的状态。为了避免进程切换时造成内核数据错误，内核在执行临界区代码时会禁止一切中断.

See <http://blog.csdn.net/deep_explore/article/details/6928228>

| **进程状态** | **说明** |
| --- | --- |
| R (TASK\_RUNNING)  可执行状态 | 只有在该状态的进程才可能在CPU上运行。而同一时刻可能有多个进程处于可执行状态，这些进程的task\_struct结构（进程控制块）被放入对应 CPU的可执行队列中（一个进程最多只能出现在一个CPU的可执行队列中）。进程调度器的任务就是从各个CPU的可执行队列中分别选择一个进程在该 CPU上运行。  很多操作系统教科书将正在CPU上执行的进程定义为RUNNING状态、而将可执行但是尚未被调度执行的进程定义为READY状态，这两种状态在linux下统一为 TASK\_RUNNING状态。 |
| S (TASK\_INTERRUPTIBLE)  可中断的睡眠状态 | 处于这个状态的进程因为等待某某事件的发生（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。  通过ps命令我们会看到，一般情况下，进程列表中的绝大多数进程都处于TASK\_INTERRUPTIBLE状态（除非机器的负载很高）。毕竟CPU就这么一两个，进程动辄几十上百个，如果不是绝大多数进程都在睡眠，CPU又怎么响应得过来。 |
| D (TASK\_UNINTERRUPTIBLE)  不可中断的睡眠状态。 | 与TASK\_INTERRUPTIBLE状态类似，进程处于睡眠状态，但是此刻进程是不可中断的。不可中断，指的并不是CPU不响应外部硬件的中断，而是指进程不响应异步信号。  绝大多数情况下，进程处在睡眠状态时，总是应该能够响应异步信号的。否则你将惊奇的发现，kill -9竟然杀不死一个正在睡眠的进程了！于是我们也很好理解，为什么ps命令看到的进程几乎不会出现TASK\_UNINTERRUPTIBLE状态，而总是 TASK\_INTERRUPTIBLE状态。  而TASK\_UNINTERRUPTIBLE状态存在的意义就在于，内核的某些处理流程是不能被打断的。如果响应异步信号，程序的执行流程中就会被插入一段用于处理异步信号的流程（这个插入的流程可能只存在于内核态，也可能延伸到用户态），于是原有的流程就被中断了。（参见《linux内核异步中断浅析》）  在进程对某些硬件进行操作时（比如进程调用read系统调用对某个设备文件进行读操作，而read系统调用最终执行到对应设备驱动的代码，并与对应的物理设备进行交互），可能需要使用TASK\_UNINTERRUPTIBLE状态对进程进行保护，以避免进程与设备交互的过程被打断，造成设备陷入不可控的状态。这种情况下的TASK\_UNINTERRUPTIBLE状态总是非常短暂的，通过ps命令基本上不可能捕捉到。 |
| T (TASK\_STOPPED or TASK\_TRACED)，  暂停状态或跟踪状态 | 向进程发送一个SIGSTOP信号，它就会因响应该信号而进入TASK\_STOPPED状态（除非该进程本身处于 TASK\_UNINTERRUPTIBLE状态而不响应信号）。（SIGSTOP与SIGKILL信号一样，是非常强制的。不允许用户进程通过 signal系列的系统调用重新设置对应的信号处理函数。）  向进程发送一个SIGCONT信号，可以让其从TASK\_STOPPED状态恢复到TASK\_RUNNING状态。  当进程正在被跟踪时，它处于TASK\_TRACED这个特殊的状态。“正在被跟踪”指的是进程暂停下来，等待跟踪它的进程对它进行操作。比如在 gdb中对被跟踪的进程下一个断点，进程在断点处停下来的时候就处于TASK\_TRACED状态。而在其他时候，被跟踪的进程还是处于前面提到的那些状态。  对于进程本身来说，TASK\_STOPPED和TASK\_TRACED状态很类似，都是表示进程暂停下来。  而TASK\_TRACED状态相当于在TASK\_STOPPED之上多了一层保护，处于TASK\_TRACED状态的进程不能响应 SIGCONT信号而被唤醒。只能等到调试进程通过ptrace系统调用执行PTRACE\_CONT、PTRACE\_DETACH等操作（通过 ptrace系统调用的参数指定操作），或调试进程退出，被调试的进程才能恢复TASK\_RUNNING状态。 |
| Z (TASK\_DEAD - EXIT\_ZOMBIE)  退出状态，进程成为僵尸进程。 | 进程在退出的过程中，处于TASK\_DEAD状态。在这个退出过程中，进程占有的所有资源将被回收，除了task\_struct结构（以及少数资源）以外。于是进程就只剩下task\_struct这么个空壳，故称为僵尸。之所以保留task\_struct，是因为task\_struct里面保存了进程的退出码、以及一些统计信息。而其父进程很可能会关心这些信息。比如在shell中，$?变量就保存了最后一个退出的前台进程的退出码，而这个退出码往往被作为if语句的判断条件。  当然，内核也可以将这些信息保存在别的地方，而将task\_struct结构释放掉，以节省一些空间。但是使用task\_struct结构更为方便，因为在内核中已经建立了从pid到task\_struct查找关系，还有进程间的父子关系。释放掉task\_struct，则需要建立一些新的数据结构，以便让父进程找到它的子进程的退出信息。  父进程可以通过wait系列的系统调用（如wait4、waitid）来等待某个或某些子进程的退出，并获取它的退出信息。然后wait系列的系统调用会顺便将子进程的尸体（task\_struct）也释放掉。子进程在退出的过程中，内核会给其父进程发送一个信号，通知父进程来“收尸”。这个信号默认是SIGCHLD，但是在通过clone系统调用创建子进程时，可以设置这个信号。 |
| X (TASK\_DEAD - EXIT\_DEAD)，  退出状态，进程即将被销毁。 | 而进程在退出过程中也可能不会保留它的task\_struct。比如这个进程是多线程程序中被detach过的进程（进程？线程？参见《linux线程浅析》）。或者父进程通过设置SIGCHLD信号的handler为SIG\_IGN，显式的忽略了SIGCHLD信号。（这是posix 的规定，尽管子进程的退出信号可以被设置为SIGCHLD以外的其他信号。）此时，进程将被置于EXIT\_DEAD退出状态，这意味着接下来的代码立即就会将该进程彻底释放。所以EXIT\_DEAD状态是非常短暂的，几乎不可能通过ps命令捕捉到。 |

#### TASK\_UNINTERRUPTIBLE状态(D)

linux系统中也存在容易捕捉的TASK\_UNINTERRUPTIBLE状态。执行vfork系统调用后，父进程将进入TASK\_UNINTERRUPTIBLE状态，直到子进程调用exit或exec（参见《神奇的vfork》）。

通过下面的代码就能得到处于TASK\_UNINTERRUPTIBLE状态的进程。

1. ＃include
2. **int**  main(**void**)
3. {
4. **if** (!vfork())
5. sleep(100);
6. }

然后我们可以试验一下TASK\_UNINTERRUPTIBLE状态的威力。不管kill还是kill -9，这个TASK\_UNINTERRUPTIBLE状态的父进程依然屹立不倒。

#### EXIT\_ZOMBIE状态(Z)

通过下面的代码能够制造一个EXIT\_ZOMBIE状态的进程：

1. #include
2. **int** main(**void**)
3. {
4. **if** (fork())
5. **while**(1);
6. sleep(100);
7. }

只要父进程不退出，这个僵尸状态的子进程就一直存在。那么如果父进程退出了呢，谁又来给子进程“收尸”？当进程退出的时候，会将它的所有子进程都托管给别的进程（使之成为别的进程的子进程）。托管给谁呢？可能是退出进程所在进程组的下一个进程（如果存在的话），或者是1号进程。所以每个进程、每时每刻都有父进程存在。除非它是1号进程。

1号进程，pid为1的进程，又称init进程。linux系统启动后，第一个被创建的用户态进程就是init进程。它有两项使命：

1. 执行系统初始化脚本，创建一系列的进程（它们都是init进程的子孙）；
2. 在一个死循环中等待其子进程的退出事件，并调用waitid系统调用来完成“收尸”工作；

init进程不会被暂停、也不会被杀死（这是由内核来保证的）。它在等待子进程退出的过程中处于TASK\_INTERRUPTIBLE状态，“收尸”过程中则处于TASK\_RUNNING状态。

#### 进程的初始状态

进程是通过fork系列的系统调用（fork、clone、vfork）来创建的，内核（或内核模块）也可以通过kernel\_thread函数创建内核进程。这些创建子进程的函数本质上都完成了相同的功能——将调用进程复制一份，得到子进程。（可以通过选项参数来决定各种资源是共享、还是私有。）那么既然调用进程处于TASK\_RUNNING状态（否则，它若不是正在运行，又怎么进行调用？），则子进程默认也处于TASK\_RUNNING状态。另外，在系统调用调用clone和内核函数kernel\_thread也接受CLONE\_STOPPED选项，从而将子进程的初始状态置为 TASK\_STOPPED。

#### 进程状态变迁

进程自创建以后，状态可能发生一系列的变化，直到进程退出。而尽管进程状态有好几种，但是进程状态的变迁却只有两个方向——从TASK\_RUNNING状态变为非TASK\_RUNNING状态、或者从非TASK\_RUNNING状态变为TASK\_RUNNING状态。

也就是说，如果给一个TASK\_INTERRUPTIBLE状态的进程发送SIGKILL信号，这个进程将先被唤醒（进入 TASK\_RUNNING状态），然后再响应SIGKILL信号而退出（变为TASK\_DEAD状态）。并不会从TASK\_INTERRUPTIBLE状态直接退出。

进程从非TASK\_RUNNING状态变为TASK\_RUNNING状态，是由别的进程（也可能是中断处理程序）执行唤醒操作来实现的。执行唤醒的进程设置被唤醒进程的状态为TASK\_RUNNING，然后将其task\_struct结构加入到某个CPU的可执行队列中。于是被唤醒的进程将有机会被调度执行。

而进程从TASK\_RUNNING状态变为非TASK\_RUNNING状态，则有两种途径：

* 响应信号而进入TASK\_STOPED状态、或TASK\_DEAD状态；
* 执行系统调用主动进入TASK\_INTERRUPTIBLE状态（如nanosleep系统调用）、或TASK\_DEAD状态（如exit 系统调用）；或由于执行系统调用需要的资源得不到满足，而进入TASK\_INTERRUPTIBLE状态或TASK\_UNINTERRUPTIBLE状态（如select系统调用）。

显然，这两种情况都只能发生在进程正在CPU上执行的情况下。

### 死锁原理

**死锁原理**

斥持夺等：互斥，持有，不可抢夺，循环等待。

### 死锁检测

TBD

### 内存泄露

See <http://www.cnblogs.com/skynet/archive/2011/02/20/1959162.html>

Windows下检查memory leak

**检查是否有内存泄露：**

* 包含头文件

|  |  |
| --- | --- |
| 头文件 | #define \_CRTDBG\_MAP\_ALLOC  #include <stdlib.h>  #include <crtdbg.h> |
| crtdbg.h | 将 malloc 和 free 函数映射到它们的调试版本，即 \_malloc\_dbg 和 \_free\_dbg |
| \_CRTDBG\_MAP\_ALLOC | 还会显示在其中分配泄漏的内存的文件。 文件名后括号中的数字（本示例中为 10）是该文件中的行号 |

* 在程序退出的地方

调用 \_CrtDumpMemoryLeaks

* 在程序开头的地方

\_CrtSetDbgFlag ( \_CRTDBG\_ALLOC\_MEM\_DF | \_CRTDBG\_LEAK\_CHECK\_DF );

**定位泄露位置：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | \_CrtMemCheckpoint( &s1 );  // memory allocations take place here  \_CrtMemCheckpoint( &s2 );    if ( \_CrtMemDifference( &s3, &s1, &s2) )     \_CrtMemDumpStatistics( &s3 ); |

 CRT 库提供一种结构类型 \_CrtMemState，您可用它存储内存状态的快照。

\_CrtMemDifference 比较两个内存状态（s1 和 s2），生成这两个状态之间差异的结果（s3）。 在程序的开始和结尾放置 \_CrtMemCheckpoint 调用，并使用\_CrtMemDifference 比较结果，是检查内存泄漏的另一种方法。 如果检测到泄漏，则可以使用 \_CrtMemCheckpoint 调用通过二进制搜索技术来划分程序和定位泄漏。

**Linux下**

Mtrace

#include <stdlib.h>

#include <mcheck.h>

int main(void) {

mtrace(); /\* Starts the recording of memory allocations and releases \*/

int\* a = NULL;

a = malloc(sizeof(int)); /\* allocate memory and assign it to the pointer \*/

if (a == NULL) {

return 1; /\* error \*/

}

free(a); /\* we free the memory we allocated so we don't have leaks \*/

muntrace();

return 0; /\* exit \*/

}

1. 编译(add -g)和运行like normal：

gcc yourProgram.c -g

./a.out

1. Memory leak information will be reported in the file specified by the MALLOC\_TRACE environment variable.this report is in computer-readable format.
2. 查看报告(调用mtrace命令查看相关报告)：

mtrace <exec\_file\_name> <malloc\_trace\_filename>

Memory not freed:

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

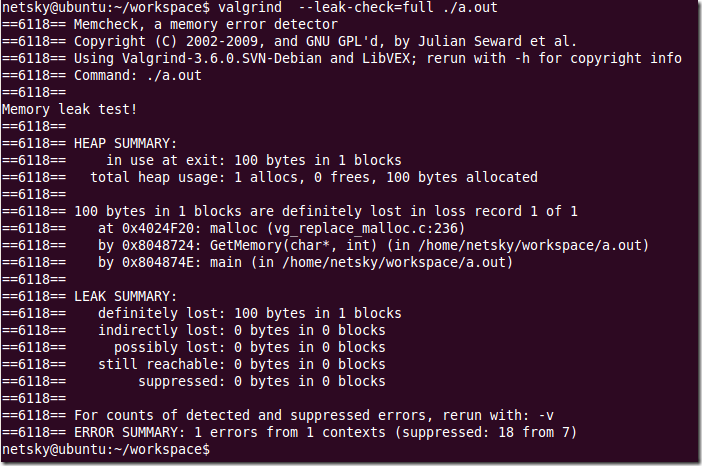
Address Size Caller

0x08049910 0x4 at /home/sureshsathiah/tips/leak.c:9

Valgrind：

1） 编译时加-g

2） 运行时: valgrind --leak-check=yes myprog arg1 arg2



### Core dump

* Core文件保存路径：

默认路径：

程序的当前工作路径，如果chdir函数改变了工作路径，则core文件保存路径会跟着变化

* 系统配置：/proc/sys/kernel/core\_pattern

默认值：core

可修改，例如：echo “/data/coredump/core”> **/proc/sys/kernel/core\_pattern**

通过修改kernel的参数，可以指定内核所生成的coredump文件的文件名。

例如，使用下面的命令使kernel生成名字为core.filename.pid格式的core dump文件：

echo “/data/coredump/core.%e.%p” >**/proc/sys/kernel/core\_pattern**

Core\_pattern的格式

可以在core\_pattern模板中使用变量还很多，见下面的列表：

%% 单个%字符

%p 所dump进程的进程ID

%u 所dump进程的实际用户ID

%g 所dump进程的实际组ID

%s 导致本次core dump的信号

%t core dump的时间 (由1970年1月1日计起的秒数)

%h 主机名

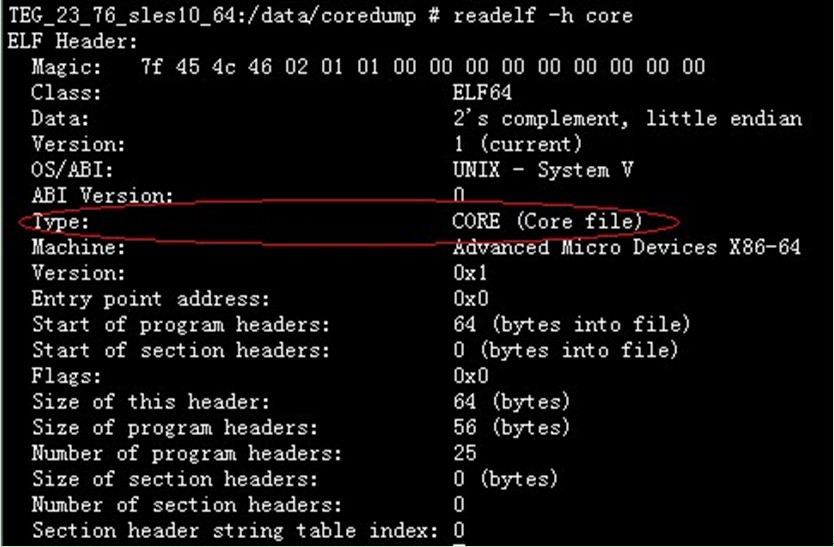
%e 程序文件名

* 添加进程ID的配置：/proc/sys/kernel/core\_uses\_pid

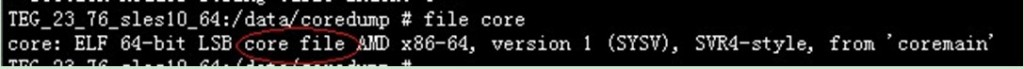
如果这个文件的内容被配置成1，那么即使core\_pattern中没有设置%p，最后生成的core dump文件名仍会加上进程ID。

* core文件类型

命令readelf



命令file



* 生成core

#ulimit –unlimited

#ulimit –c [size] //设置大小

Ulimit只对当前会话有效，若想系统均有效

在/etc/profile中加入以下一行，这将允许生成coredump文件

ulimit-c unlimited

在rc.local中加入以下一行，这将使程序崩溃时生成的coredump文件位于/data/coredump/目录下:

echo /data/coredump/core.%e.%p> /proc/sys/kernel/core\_pattern

注意rc.local在不同的环境，存储的目录可能不同，susu下可能在/etc/rc.d/rc.local

The core file will not be generated if

(a) the process was set-user-ID and the current user is not the owner of the program file, or

(b) the process was set-group-ID and the current user is not the group owner of the file,

(c) the user does not have permission to write in the current working directory,

(d) the file already exists and the user does not have permission to write to it, or

(e) the file is too big (recall the RLIMIT\_CORE limit in Section 7.11). The permissions of the core file (assuming that the file doesn't already exist) are usually user-read and user-write, although Mac OS X sets only user-read.

* 生成core文件的错误类型

内存越界，多线程使用了线程不安全的函数，多线程读写数据未加锁，非法指针，堆栈溢出等等

* gdb定位

#gdb [exec file] [core file]

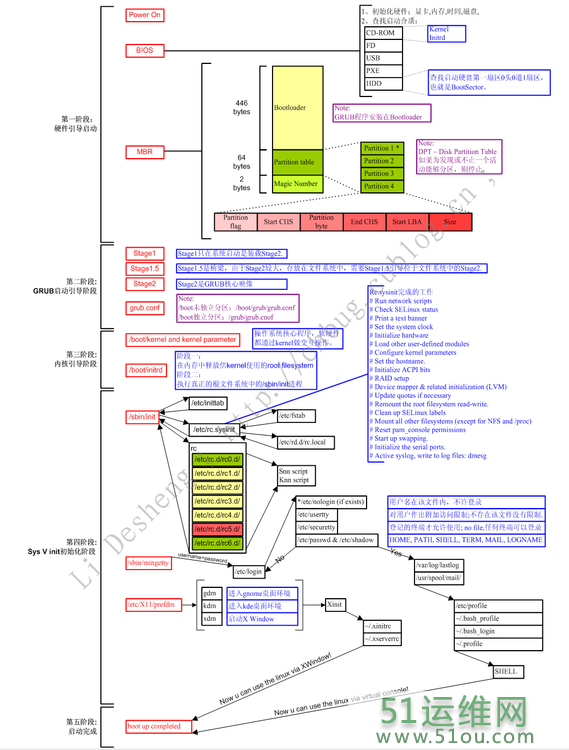
#where //跳到错误的地方

#bt //查看调用栈

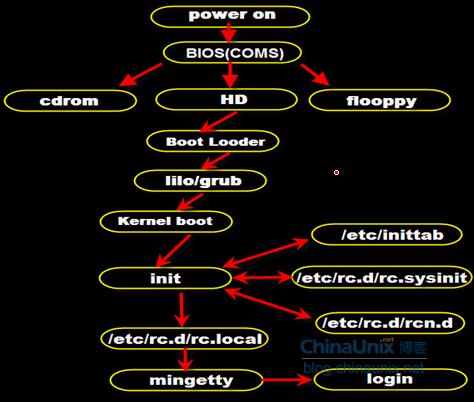
## linux系统启动过程

see <http://blog.chinaunix.net/uid-23069658-id-3142047.html>





http://blog.chinaunix.net/uid-26495963-id-3066282.html



| **step** | **说明** |
| --- | --- |
| 1 加载BIOS | 当你打开计算机电源，计算机会首先加载BIOS信息，BIOS信息是如此的重要，以至于计算机必须在最开始就找到它。这是因为BIOS中包含了CPU的相关信息、设备启动顺序信息、硬盘信息、内存信息、时钟信息、PnP特性等等。在此之后，计算机心里就有谱了，知道应该去读取哪个硬件设备了。 |
| 2读取MBR | 众所周知，硬盘上第0磁道第一个扇区被称为MBR，也就是Master Boot Record，即主引导记录，它的大小是512字节，别看地方不大，可里面却存放了预启动信息、分区表信息。  系统找到BIOS所指定的硬盘的MBR后，就会将其复制到0×7c00地址所在的物理内存中。其实被复制到物理内存的内容就是Boot Loader，而具体到你的电脑，那就是lilo或者grub了。 |
| 3 boot loader | Boot Loader 就是在操作系统内核运行之前运行的一段小程序。通过这段小程序，我们可以初始化硬件设备、建立内存空间的映射图，从而将系统的软硬件环境带到一个合适的状态，以便为最终调用操作系统内核做好一切准备。  Boot Loader有若干种，其中Grub、Lilo和spfdisk是常见的Loader。  我们以Grub为例来讲解吧，毕竟用lilo和spfdisk的人并不多。  系统读取内存中的grub配置信息（一般为menu.lst或grub.lst），并依照此配置信息来启动不同的操作系统。 |
| 4加载内核 | 根据grub设定的内核映像所在路径，系统读取内存映像，并进行解压缩操作。此时，屏幕一般会输出“Uncompressing Linux”的提示。当解压缩内核完成后，屏幕输出“OK, booting the kernel”。  系统将解压后的内核放置在内存之中，并调用start\_kernel()函数来启动一系列的初始化函数并初始化各种设备，完成Linux核心环境的建立。至此，Linux内核已经建立起来了，基于Linux的程序应该可以正常运行了。 |
| 5用户层init依据inittab文件来设定运行等级 | 内核被加载后，第一个运行的程序便是/sbin/init，该文件会读取/etc/inittab文件，并依据此文件来进行初始化工作。  其实/etc/inittab文件最主要的作用就是设定Linux的运行等级，其设定形式是“：id:5:initdefault:”，这就表明Linux需要运行在等级5上。Linux的运行等级设定如下：  0：关机  1：单用户模式  2：无网络支持的多用户模式  3：有网络支持的多用户模式  4：保留，未使用  5：有网络支持有X-Window支持的多用户模式  6：重新引导系统，即重启  关于/etc/inittab文件的学问，其实还有很多 |
| 6init进程执行rc.sysinit | 在设定了运行等级后，Linux系统执行的第一个用户层文件就是/etc/rc.d/rc.sysinit脚本程序，它做的工作非常多，包括设定PATH、设定网络配置（/etc/sysconfig/network）、启动swap分区、设定/proc等等。如果你有兴趣，可以到/etc/rc.d中查看一下rc.sysinit文件，里面的脚本够你看几天的 |
| 7启动内核模块 | 具体是依据/etc/modules.conf文件或/etc/modules.d目录下的文件来装载内核模块。 |
| 8执行不同运行级别的脚本程序 | 根据运行级别的不同，系统会运行rc0.d到rc6.d中的相应的脚本程序，来完成相应的初始化工作和启动相应的服务。 |
| 9执行/etc/rc.d/rc.local | 你如果打开了此文件，里面有一句话，读过之后，你就会对此命令的作用一目了然：  # This script will be executed \*after\* all the other init scripts.  # You can put your own initialization stuff in here if you don’t  # want to do the full Sys V style init stuff.  rc.local就是在一切初始化工作后，Linux留给用户进行个性化的地方。你可以把你想设置和启动的东西放到这里。 |
| 10执行/bin/login程序，进入登录状态 | 此时，系统已经进入到了等待用户输入username和password的时候了，你已经可以用自己的帐号登入系统了。:) |

## Linux run level

from /etc/inittab

9 # Default runlevel. The runlevels used by RHS are:

10 # 0 - halt (Do NOT set initdefault to this)

11 # 1 - Single user mode

12 # 2 - Multiuser, without NFS (The same as 3, if you do not have networking)

13 # 3 - Full multiuser mode

14 # 4 - unused

15 # 5 - X11

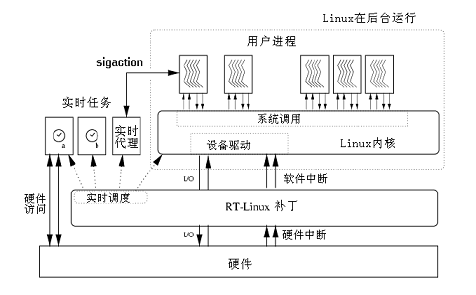
16 # 6 - reboot (Do NOT set initdefault to this)

## RTLinux

See <http://bbs.chinaunix.net/thread-2007414-1-1.html>

### 实现原理

在Linux 操作系统中,调度算法（其于最大吞吐量准则）、设备驱动、不可中断的系统调用、中断屏蔽以及虚拟内存的使用等因素,都会导致系统在时间上的不可预测性,决定了Linux操作系统不能处理硬实时任务。RTLinux为避免这些问题,在Linux内核与硬件之间增加了一个虚拟层（通常称作虚拟机）,构筑了一个小的、时间上可预测的、与Linux内核分开的实时内核,使得在其中运行的实时进程满足硬实时性。并且RTLinux和Linux构成一个完备的整体,能够完成既包括实时部分又包括非实时部分的复杂任务。



### 硬实时性

RTLinux 将Linux源码中所有的cli、sti、iret指令分别用宏S\_CLI、S\_STI、S\_IRET替换,引入虚拟层将截取所有的硬件中断,分割 Linux系统与硬件中断之间的直接联系。

|  |  |
| --- | --- |
| 当RTLinux虚拟层接收到与实时处理有关的硬件中断时 | 立即启动执行相应的实时中断服务程序 |
| 接收到与实时处理无关的中断时 | 先保存相应的信息,等到RTLinux内核空闲时通过软中断传递给Linux内核去处理 |

这样就使得RTLinux内核不受各种软、硬件中断的影响,不会造成时间上的不可预测性。

|  |  |  |
| --- | --- | --- |
|  | Linux | RTLinux |
| 调度策略 | 最大吞吐量准则 | 优先级  同时支持：  最短时限最先调度（EDP）、确定周期调度（RM）（周期短的实时任务具有高的优先级） |

### 完备性

为了满足当今实时应用程序的多种需求,通常采用在实时控制内核上增加这些服务或完全修改标准操作系统内核的方法,而RTLinux所采用的是一种新型高效的方式。将一个简单的小型实时内核与Linux内核共存,用简单的小型实时内核处理实时任务,将非实时任务交给Linux内核去处理,而 Linux内核本身也作为一个RTLinux实时内核在空闲时运行的进程。这种将实时系统和平均时间优化的标准Linux操作系统协同工作的方式,使得许多实时应用都显示出一种增效。

实时内核中的实时任务可以直接访问硬件,不使用虚拟内存,给实时进程提供了很大的灵活性;

运行在Linux用户空间中的非实时任务,可以方便地使用系统提供的各种资源（网络、文件系统等）,并受到系统的保护,增加了系统的安全性。

### RTLinux的主要功能

See <http://www.cnblogs.com/cubean/archive/2010/03/23/1692609.html>

##### a.中断仿真

在中断控制硬件与LINUX核心之间放置一个软件仿真层。具体做法是，在LINUX源码中出现cli、sti和 iret的所有地方都用仿真宏：S\_CLI、S\_STI和S\_IRET来替换。所有的硬件中断就都被仿真器所截获。

当需要关中断时，就将仿真器中的一个变量置0。不论何时若有中断发生，仿真器就检查这个变量。如果是1（LINUX 已开中断），就立即调用LINUX的中断处理程序；否则，LINUX中断被禁止，中断处理程序不会被调用，而是在保存着所有挂起中断的信息的变量的相应位 置1。当LINUX重新开中断，所有挂起中断的处理程序都会被执行。这种仿真方式可以称之为"软中断"。

##### b.实时任务

实时任务是在一个由核心控制的调度程序的调度下执行的用户定义的程序。

RT-LINUX最初将实时任务设计成ELF格式的目标文件。这一设计方案的最大缺点就是性能比较差。原因在于，第 一，486的缓存是虚拟的。所以每当页表目录的基址寄存器改变时，TLB（转换后备缓冲器）就会失效。由于实时任务的上下文转换频繁，所以TLB的频繁失 效就导致系统性能的严重下降。第二，486的保护级别变换耗时不少。比如，陷入更高级别时需要71个循环，而其它指令一般少于10个循环。

解决的办法就是使用可加载模组技术，所有的实时任务都同处于一个地址空间-内核地址空间，不仅避免了频繁的TLB失 效，同时也消除了变换保护级别的消耗，而且任务转换也变得相当容易。

##### c.进程调度

实时系统的进程调度的主要任务就是满足实时任务在时间上的要求。调度算法的种类很多，没有一个策略是放之四海而皆准的，因此采用哪种算法要取决于具体应用。

RT-LINUX采用的方法是允许用户编写自己的调度程序，并可以编译成模组的形式。这样就可以方便地试验不同的策 略和算法对于某一特定应用的适合性，从中选出最优。

RT-LINUX自带的是一个基于优先数的抢占式调度程序。此调度程序将LINUX当作具有最低优先数的实时任务。 因此，LINUX只在实时系统无任何实时任务是才运行。在从LINUX切换到实时任务时，系统记下软中断的状态并禁止软中断。在切换回来实，再恢复软中断 的状态。

##### d.时钟

调度程序需要精确的时钟才能准确操作。调度通常是在特定的时刻进行任务切换。时钟的偏差会引起预定调度的偏差，导致 产生被称为任务发布抖动的现象。这是一种应该尽量避免的不良现象。

RT-LINUX的解决办法是，将IBM PC兼容机中的时钟芯片Intel 8254设置为中断开启终端计数模式。在这种模式下，精度可以达到1毫秒。这样在降低中断处理的影响的同时，获得了较高的时钟精度。

##### e.IPC

由于标准LINUX核心可以被实时任务在任意时刻抢占，所以实时任务无法安全地调用LINUX的程序。但是总要有一 个信息交换的机制。

在RT-LINUX中所用的信息交换方式是RT-FIFO（实时队列）。它与UNIX的管道非常相似，都是一个无结构的数据流。通过RT-FIFO，LINUX的进程之间，实时进程之间，以及LINUX的核心与实时进程之间可以交换信息。

对于一个普通的进程来说，RT-FIFO就是一个特殊的字符文件。这些文件必须自建。

### 打RT补丁

See <http://bbs.chinaunix.net/thread-2083398-1-1.html>

/\*

\* 安装和添加实时补丁

\* 1. RedHat Linux Installation

\* If you have not done it before, make a separate disk partition and install

\* RedHat linux on the partition. RedHat 8.0 (Psyche) is recommended since

\* it is most compatible with the RT-Linux we will use in this course. Also,

\* make sure to install kernel development tools, gcc compiler, and utilities

\* such as patch, depmod, make, and bzip2.

\* 2. Get linux kernel

\* Download \linux-2.4.18.tar.bz2" from the course website.

\* 3. Get rtlinux kernel and patches

\* Download \rtlinux-3.2-pre1.tar.bz2" from the course website.

\* 4. Put a fresh copy of the rtlinux kernel in /usr/src/rtlinux-3.2-pre1

\* Use the following commands.

\* cd /usr/src

\* tar xjf rtlinux-3.2-pre1.tar.bz2

\* This will create \rtlinux-3.2-pre1" directory under /usr/src.

\* 5. Put a fresh copy of the linux kernel in /usr/src/rtlinux-3.2-pre1/linux

\* Use the following commands.

\* cd /usr/src/rtlinux-3.2-pre1

\* tar xjf linux-2.4.18.tar.bz2

\* This will create \linux" directory under /usr/src/rtlinux-3.2-pre1.

\* 6. Patch the linux kernel with the rtlinux patch

\* cd /usr/src/rtlinux-3.2-pre1/patches

\* bzip2 -d kernel patch-2.4.18-rtl3.2-pre1.bz2

\* cd /usr/src/rtlinux-3.2-pre1/linux

\* patch -p1 < /usr/src/rtlinux-3.2-pre1/patches/kernel patch-2.4.18

\* 7. Clean all "\.o" files and stale dependencies

\* cd /usr/src/rtlinux-3.2-pre1/linux

\* make mrproper

\* 8. Configure the linux kernel

\* cd /usr/src/rtlinux-3.2-pre1/linux

\* make config (text mode) or

\* make xconfig (X mode)

\* 9. Build a new linux kernel

\* Use the following commands in order.

\* cd /usr/src/rtlinux-3.2-pre1/linux

\* make dep

\* make bzImage

\* make modules

\* su

\* make modules install

\* cp arch/i386/boot/bzImage /boot/rtzImage

\* 10. Configure your boot loader

\* In the following explanation, we assume that the root file system \/" is

\* mapped to /dev/hda3 and the boot file system \/boot" is mapped to /dev/hda2.

\* You can check out your mapping using \df" command. If you are using LILO

\* as your boot manager, add the following lines to the file \/etc/lilo.conf".

\* image = /boot/rtzImage

\* label = rtlinux

\* read-only

\* root = /dev/hda3

\* The /dev/hda3 should be the device on which your root file system has been

\* installed. Then, use the following command.

\* /sbin/lilo

\* For more details for your specific setting,

\* see [url]http://www.freeos.com/articles/2701/[/url] and

\* [url]http://www.linuxheadquarters.com/howto/basic/lilo.shtml.[/url]

\* If your are using GRUB as your boot manager, add the following lines to the file \/etc/grub.conf".

\* title rtlinux

\* root (hd0, 1)

\* kernel /rtzImage ro root = /dev/hda3

\* The /dev/hda3 should be the device on which your root file system has been

\* installed. (hd0,1) corresponds to the first (0) physical disk derive's

\* second (1) partition, which is /dev/hda2. It should be the partition where

\* the boot file system \/boot" resides. For more details for

\* your specific setting, see [url]http://www.gnu.org/software/grub/.[/url]

\* 11. Reboot and select rtlinux from boot image options

\* RTLinux should boot.

\* 12. Configure RTLinux

\* cd /usr/src/rtlinux-3.2-pre1

\* make xconfig (accept default)

\* 13. Compile RTLinux Use the following commands in order.

\* cd /usr/src/rtlinux-3.2-pre1

\* make

\* su (become the root to do the followings)

\* make devices

\* make install

\* 14. Reboot and select rtlinux from boot image options

\*/

/\*

\* 运行实时内核

\* First of all, you should be the root to do the followings.

\* In order to run rtlinux applications given in \examples" directory,

\* you first insert dynamic rtlinux kernel modules like mbuff, rtl\_fifo,

\* rtl, rtl posixio, rtl sched, and rtl time. You can insert each module

\* with \insmod modulename.o". Fortunately, the script \rtlinux" can do this

\* for you at a single step. Try

\* rtlinux status ( look at which module has been loaded )

\* rtlinux start ( insert all rtlinux modules )

\* rtlinux status ( check whether all rtlinux modules are loaded successfully )

\* If you can see all the modules loaded, you are done. Otherwise, you might see

\* error messages while you were doing \rtlinux start". This may be due to incorrect

\* configuration of linux kernel.

\* 运行实时内核模块

\* insmod hello.o 载入模块

\* lsmod 查看已经载入的模块

\*/

## Windows

# Linux常用工具

## ps

|  |  |
| --- | --- |
| UNIX option | Must be preceded with dash |
| BSD option | Must not be preceded with dash |
| GNU long option | Must be preceded with two dashes |

|  |  |
| --- | --- |
| Option type | Option detail |
| SIMPLE PROCESS SELECTION | a Lift the BSD-style "only yourself" restriction, which is imposed upon the set of all processes when some BSD-style (without "-") options are used or when the ps personality setting is BSD-like. The set of processes  selected in this manner is in addition to the set of processes selected by other means. An alternate description is that this option causes ps to list all processes with a terminal (tty), or to list all processes  when used together with the x option.  -A Select all processes. Identical to -e.  -a Select all processes except both session leaders (see getsid(2)) and processes not associated with a terminal.  -d Select all processes except session leaders.  --deselect  Select all processes except those that fulfill the specified conditions (negates the selection). Identical to -N.  -e Select all processes. Identical to -A.  g Really all, even session leaders. This flag is obsolete and may be discontinued in a future release. It is normally implied by the a flag, and is only useful when operating in the sunos4 personality.  -N Select all processes except those that fulfill the specified conditions (negates the selection). Identical to --deselect.  T Select all processes associated with this terminal. Identical to the t option without any argument.  r Restrict the selection to only running processes.  x Lift the BSD-style "must have a tty" restriction, which is imposed upon the set of all processes when some BSD-style (without "-") options are used or when the ps personality setting is BSD-like. The set of processes  selected in this manner is in addition to the set of processes selected by other means. An alternate description is that this option causes ps to list all processes owned by you (same EUID as ps), or to list all  processes when used together with the a option. |
| PROCESS SELECTION BY LIST | These options accept a single argument in the form of a blank-separated or comma-separated list. They can be used multiple times. For example: ps -p "1 2" -p 3,4    -123 Identical to --pid 123.    123 Identical to --pid 123.    -C cmdlist  Select by command name. This selects the processes whose executable name is given in cmdlist.    -G grplist  Select by real group ID (RGID) or name. This selects the processes whose real group name or ID is in the grplist list. The real group ID identifies the group of the user who created the process, see getgid(2).    -g grplist  Select by session OR by effective group name. Selection by session is specified by many standards, but selection by effective group is the logical behavior that several other operating systems use. This ps will  select by session when the list is completely numeric (as sessions are). Group ID numbers will work only when some group names are also specified. See the -s and --group options.    --Group grplist  Select by real group ID (RGID) or name. Identical to -G.    --group grplist  Select by effective group ID (EGID) or name. This selects the processes whose effective group name or ID is in grplist. The effective group ID describes the group whose file access permissions are used by the  process (see getegid(2)). The -g option is often an alternative to --group.    p pidlist  Select by process ID. Identical to -p and --pid.    -p pidlist  Select by PID. This selects the processes whose process ID numbers appear in pidlist. Identical to p and --pid.    --pid pidlist  Select by process ID. Identical to -p and p.    --ppid pidlist  Select by parent process ID. This selects the processes with a parent process ID in pidlist. That is, it selects processes that are children of those listed in pidlist.    q pidlist  Select by process ID (quick mode). Identical to -q and --quick-pid.    -q pidlist  Select by PID (quick mode). This selects the processes whose process ID numbers appear in pidlist. With this option ps reads the necessary info only for the pids listed in the pidlist and doesn't apply additional  filtering rules. The order of pids is unsorted and preserved. No additional selection options, sorting and forest type listings are allowed in this mode. Identical to q and --quick-pid.    --quick-pid pidlist  Select by process ID (quick mode). Identical to -q and q.    -s sesslist  Select by session ID. This selects the processes with a session ID specified in sesslist.    --sid sesslist  Select by session ID. Identical to -s.    t ttylist  Select by tty. Nearly identical to -t and --tty, but can also be used with an empty ttylist to indicate the terminal associated with ps. Using the T option is considered cleaner than using t with an empty ttylist. |
| OUTPUT FORMAT CONTROL | These options are used to choose the information displayed by ps. The output may differ by personality.    -c Show different scheduler information for the -l option.    --context  Display security context format (for SELinux).    -f Do full-format listing. This option can be combined with many other UNIX-style options to add additional columns. It also causes the command arguments to be printed. When used with -L, the NLWP (number of threads)  and LWP (thread ID) columns will be added. See the c option, the format keyword args, and the format keyword comm.    -F Extra full format. See the -f option, which -F implies.    --format format  user-defined format. Identical to -o and o.    j BSD job control format.    -j Jobs format.    l Display BSD long format.    -l Long format. The -y option is often useful with this.    -M Add a column of security data. Identical to Z (for SELinux).    O format  is preloaded o (overloaded). The BSD O option can act like -O (user-defined output format with some common fields predefined) or can be used to specify sort order. Heuristics are used to determine the behavior of  this option. To ensure that the desired behavior is obtained (sorting or formatting), specify the option in some other way (e.g. with -O or --sort). When used as a formatting option, it is identical to -O, with the  BSD personality.    -O format  Like -o, but preloaded with some default columns. Identical to -o pid,format,state,tname,time,command or -o pid,format,tname,time,cmd, see -o below.    o format  Specify user-defined format. Identical to -o and --format.    -o format  User-defined format. format is a single argument in the form of a blank-separated or comma-separated list, which offers a way to specify individual output columns. The recognized keywords are described in the  STANDARD FORMAT SPECIFIERS section below. Headers may be renamed (ps -o pid,ruser=RealUser -o comm=Command) as desired. If all column headers are empty (ps -o pid= -o comm=) then the header line will not be output.  Column width will increase as needed for wide headers; this may be used to widen up columns such as WCHAN (ps -o pid,wchan=WIDE-WCHAN-COLUMN -o comm). Explicit width control (ps opid,wchan:42,cmd) is offered too.  The behavior of ps -o pid=X,comm=Y varies with personality; output may be one column named "X,comm=Y" or two columns named "X" and "Y". Use multiple -o options when in doubt. Use the PS\_FORMAT environment variable  to specify a default as desired; DefSysV and DefBSD are macros that may be used to choose the default UNIX or BSD columns.    s Display signal format.    u Display user-oriented format.    v Display virtual memory format.    X Register format.    -y Do not show flags; show rss in place of addr. This option can only be used with -l.  Z Add a column of security data. Identical to -M (for SELinux). |
| OUTPUT MODIFIERS | c Show the true command name. This is derived from the name of the executable file, rather than from the argv value. Command arguments and any modifications to them are thus not shown. This option effectively turns  the args format keyword into the comm format keyword; it is useful with the -f format option and with the various BSD-style format options, which all normally display the command arguments. See the -f option, the  format keyword args, and the format keyword comm.    --cols n  Set screen width.    --columns n  Set screen width.    --cumulative  Include some dead child process data (as a sum with the parent).    e Show the environment after the command.    f ASCII art process hierarchy (forest).    --forest  ASCII art process tree.    h No header. (or, one header per screen in the BSD personality). The h option is problematic. Standard BSD ps uses this option to print a header on each page of output, but older Linux ps uses this option to totally  disable the header. This version of ps follows the Linux usage of not printing the header unless the BSD personality has been selected, in which case it prints a header on each page of output. Regardless of the  current personality, you can use the long options --headers and --no-headers to enable printing headers each page or disable headers entirely, respectively.    -H Show process hierarchy (forest).    --headers  Repeat header lines, one per page of output.    k spec Specify sorting order. Sorting syntax is [+|-]key[,[+|-]key[,...]]. Choose a multi-letter key from the STANDARD FORMAT SPECIFIERS section. The "+" is optional since default direction is increasing numerical or  lexicographic order. Identical to --sort.    Examples:  ps jaxkuid,-ppid,+pid  ps axk comm o comm,args  ps kstart\_time -ef    --lines n  Set screen height.    -n namelist  Set namelist file. Identical to N. The namelist file is needed for a proper WCHAN display, and must match the current Linux kernel exactly for correct output. Without this option, the default search path for the  namelist is:    $PS\_SYSMAP  $PS\_SYSTEM\_MAP  /proc/\*/wchan  /boot/System.map-$(uname -r)  /boot/System.map  /lib/modules/$(uname -r)/System.map  /usr/src/linux/System.map  /System.map    n Numeric output for WCHAN and USER (including all types of UID and GID).    N namelist  Specify namelist file. Identical to -n, see -n above.    --no-headers  Print no header line at all. --no-heading is an alias for this option.    O order  Sorting order (overloaded). The BSD O option can act like -O (user-defined output format with some common fields predefined) or can be used to specify sort order. Heuristics are used to determine the behavior of  this option. To ensure that the desired behavior is obtained (sorting or formatting), specify the option in some other way (e.g. with -O or --sort).    For sorting, obsolete BSD O option syntax is O[+|-]k1[,[+|-]k2[,...]]. It orders the processes listing according to the multilevel sort specified by the sequence of one-letter short keys k1,k2, ... described in the  OBSOLETE SORT KEYS section below. The "+" is currently optional, merely re-iterating the default direction on a key, but may help to distinguish an O sort from an O format. The "-" reverses direction only on the key  it precedes.    --rows n  Set screen height.    S Sum up some information, such as CPU usage, from dead child processes into their parent. This is useful for examining a system where a parent process repeatedly forks off short-lived children to do work.    --sort spec  Specify sorting order. Sorting syntax is [+|-]key[,[+|-]key[,...]]. Choose a multi-letter key from the STANDARD FORMAT SPECIFIERS section. The "+" is optional since default direction is increasing numerical or  lexicographic order. Identical to k. For example: ps jax --sort=uid,-ppid,+pid    w Wide output. Use this option twice for unlimited width.    -w Wide output. Use this option twice for unlimited width.    --width n  Set screen width. |
| THREAD DISPLAY | H Show threads as if they were processes.  -L Show threads, possibly with LWP and NLWP columns.  m Show threads after processes.  -m Show threads after processes.  -T Show threads, possibly with SPID column. |
| OTHER INFORMATION | --help section Print a help message. The section argument can be one of simple, list, output, threads, misc or all. The argument can be shortened to one of the underlined letters as in: s|l|o|t|m|a.  --info Print debugging info.  L List all format specifiers.  V Print the procps-ng version.  -V Print the procps-ng version.  --version  Print the procps-ng version. |

NOTES

|  |  |
| --- | --- |
| NOTES | This ps works by reading the virtual files in /proc. This ps does not need to be setuid kmem or have any privileges to run. Do not give this ps any special permissions.    This ps needs access to namelist data for proper WCHAN display. For kernels prior to 2.6, the System.map file must be installed.    CPU usage is currently expressed as the percentage of time spent running during the entire lifetime of a process. This is not ideal, and it does not conform to the standards that ps otherwise conforms to. CPU usage is  unlikely to add up to exactly 100%.    The SIZE and RSS fields don't count some parts of a process including the page tables, kernel stack, struct thread\_info, and struct task\_struct. This is usually at least 20 KiB of memory that is always resident. SIZE is  the virtual size of the process (code+data+stack).    Processes marked <defunct> are dead processes (so-called "zombies") that remain because their parent has not destroyed them properly. These processes will be destroyed by init(8) if the parent process exits.    If the length of the username is greater than the length of the display column, the numeric user ID is displayed instead.    Commands options such as ps -aux are not recommended as it is a confusion of two different standards. According to the POSIX and UNIX standards, the above command asks to display all processes with a TTY (generally the  commands users are running) plus all processes owned by a user named "x". If that user doesn't exist, then ps will assume you really meant "ps aux". |
| PROCESS FLAGS | The sum of these values is displayed in the "F" column, which is provided by the flags output specifier:    1 forked but didn't exec  4 used super-user privileges |
| PROCESS STATE CODES | Here are the different values that the s, stat and state output specifiers (header "STAT" or "S") will display to describe the state of a process:    D uninterruptible sleep (usually IO)  R running or runnable (on run queue)  S interruptible sleep (waiting for an event to complete)  T stopped by job control signal  t stopped by debugger during the tracing  W paging (not valid since the 2.6.xx kernel)  X dead (should never be seen)  Z defunct ("zombie") process, terminated but not reaped by its parent    For BSD formats and when the stat keyword is used, additional characters may be displayed:    < high-priority (not nice to other users)  N low-priority (nice to other users)  L has pages locked into memory (for real-time and custom IO)  s is a session leader  l is multi-threaded (using CLONE\_THREAD, like NPTL pthreads do)  + is in the foreground process group |

## umask

**用户创建一个文件（或目录）后，该文件（或目录）有其默认的访问权限**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **默认权限** | **umask默认值** | **实际权限** |
| **文件** | **666** | **022** | **644** |
| **目录** | **777** |  | **755** |

**所以：**

**实际权限 = 默认权限-umask**

**$ umask**

**0022**

**$ ls -l**

**-rw-r--r-- 1 limwu linseeusers\_asb\_dhn\_beijing 0 Aug 15 08:01 aa**

**drwxr-xr-x 2 limwu linseeusers\_asb\_dhn\_beijing 4096 Aug 15 08:01 dd**

**权限说明**

所有着、用户组或其他用户的读、写及执行权限就构成了一个有9种类型的权限组

| **位** | **说明** |
| --- | --- |
| 1 | d：表示是一个目录，事实上在ext2fs中，目录是一个特殊的文件。  －：表示这是一个普通的文件。  l: 表示这是一个符号链接文件，实际上它指向另一个文件。  b、c：分别表示区块设备和其他的外围设备，是特殊类型的文件。  s、p：这些文件关系到系统的数据结构和管道，通常很少见到。 |
| 2-4 | 所有者的写，读，执行权限 |
| 5-7 | 用户组的写，读，执行权限 |
| 8-10 | 其他用户的写，读，执行权限 |

**特殊权限**

1. s或S（SUID,Set UID）：

可执行的文件搭配这个权限，便能得到特权，任意存取该文件的所有者能使用的全部系统资源。请注意具备SUID权限的文件，黑客经常利用这种权限，以SUID配上root帐号拥有者，无声无息地在系统中开扇后门，供日后进出使用。

1. s或S（SGID,Set GID）：

设置在文件上面，其效果与SUID相同，只不过将文件所有者换成用户组，该文件就可以任意存取整个用户组所能使用的系统资源。

1. T或T（Sticky）：

/tmp和 /var/tmp目录供所有用户暂时存取文件，亦即每位用户皆拥有完整的权限进入该目录，去浏览、删除和移动文件。

因为SUID、SGID、Sticky占用x的位置来表示，所以在表示上会有大小写之分。

|  |  |
| --- | --- |
| 同时开启执行权限和SUID、SGID、Sticky | 则权限表示字符是小写的  -rwsr-sr-t 1 root root 4096 6月 23 08：17 conf |
| 如果关闭执行权限 | 则表示字符会变成大写：  -rwSr-Sr-T 1 root root 4096 6月 23 08：17 conf |

**$ ll -d /tmp**

**drwxrwxrwt 7 root root 4096 Aug 15 08:00 /tmp**

## fuser

See <https://www.digitalocean.com/community/tutorials/how-to-use-the-linux-fuser-command>

Show which processes use the named files, sockets, or filesystems.

How To View Processes Using A Directory

# fuser -v .

What process is using your tcp or udp socket?

root@exampleuser-X55CR:~# nc -l -p 80

Since a tcp server is listening on port 80, the fuser utility can be used to find the process which is using the server’s socket. The -v option is used to put the fuser utility in verbose mode and the -n option is used to select the tcp protocol as a name space.

In the following command, the -k option is used to kill the process which is using the tcp listener running on port 123. To make sure that the user does not kill a wrong process, the -i option is used which asks the user for confirmation before killing a process.

root@exampleuser-X55CR:~# fuser -k 123/tcp

123/tcp: 11543

root@exampleuser-X55CR:~# fuser -i -k 123/tcp

123/tcp: 12216

Kill process 12216 ? (y/N)

Use The -6 Option To Look For IPv6 Sockets.

the -l option helps to find the list of signals that can be used with the fuser tool.

root@exampleuser-X55CR:~# fuser -l

HUP INT QUIT ILL TRAP ABRT IOT BUS FPE KILL USR1 SEGV USR2 PIPE ALRM TERM

STKFLT CHLD CONT STOP TSTP TTIN TTOU URG XCPU XFSZ VTALRM PROF WINCH IO PWR SYS

UNUSED

Usage

Usage: fuser [ -a | -s | -c ] [ -n SPACE ] [ -SIGNAL ] [ -kimuv ] NAME...

[ - ] [ -n SPACE ] [ -SIGNAL ] [ -kimuv ] NAME...

fuser -l

fuser -V

Show which processes use the named files, sockets, or filesystems.

-a display unused files too

-c mounted FS

-f silently ignored (for POSIX compatibility)

-i ask before killing (ignored without -k)

-k kill processes accessing the named file

-l list available signal names

-m show all processes using the named filesystems

-n SPACE search in this name space (file, udp, or tcp)

-s silent operation

-SIGNAL send this signal instead of SIGKILL

-u display user IDs

-v verbose output

-V display version information

-4 search IPv4 sockets only

-6 search IPv6 sockets only

- reset options

udp/tcp names: [local\_port][,[rmt\_host][,[rmt\_port]]]

## mount

See <http://www.cnblogs.com/xd502djj/p/3809375.html>

格式：mount [-参数] [设备名称] [挂载点]

参数：

-a 安装在/etc/fstab文件中类出的所有文件系统。

-f 伪装mount，作出检查设备和目录的样子，但并不真正挂载文件系统。

-n 不把安装记录在/etc/mtab 文件中。

-r 讲文件系统安装为只读。

-v 详细显示安装信息。

-w 将文件系统安装为可写，为命令默认情况。

-t 指定设备的文件系统类型，常见的有：

ext2 linux目前常用的文件系统

msdos MS-DOS的fat，就是fat16

vfat windows98常用的fat32

nfs 网络文件系统

iso9660 CD-ROM光盘标准文件系统

ntfs windows NT/2000/XP的文件系统

auto 自动检测文件系统

-o 指定挂载文件系统时的选项，有些也可写到在/etc/fstab中。常用的有：

defaults 使用所有选项的默认值（auto、nouser、rw、suid）

auto/noauto 允许/不允许以 –a选项进行安装

dev/nodev 对/不对文件系统上的特殊设备进行解释

exec/noexec 允许/不允许执行二进制代码

suid/nosuid 确认/不确认suid和sgid位

user /nouser 允许/不允许一般用户挂载

codepage=XXX 代码页

iocharset=XXX 字符集

ro 以只读方式挂载

rw 以读写方式挂载

remount 重新安装已经安装了的文件系统

loop 挂载回旋设备

需要注意的是，挂载点必须是一个已经存在的目录，这个目录可以不为空，但挂载后这个目录下以前的内容将不可用，umount以后会恢复正常。使用多个-o参数的时候，-o 只用一次，参数之间用半角逗号隔开：

CODE:

# mount –o remount,rw /

例如要挂载windows下文件系统为FAT32的D盘，一般而言在Linux下这个分区对应/dev/hda5，根据具体的分区情况会有不同，这里就以hda5来举例说明：

CODE:

# mkdir /mnt/hda5 //创建hda5的目录作为挂载点，位置和目录名可自定义//

# mount -t vfat /dev/hda5 /mnt/hda5

一般而言，Linux会自动探测分区的文件系统，除非让你指定时，否则-t vfat 可以省掉。

CODE:

# mount /dev/hda5 /mnt/hda5

这样就可以进入/mnt/hda5目录去访问分区中的资源了。

| **常见问题** | **命令** |
| --- | --- |
| 显示不了中文文件为问号/乱码 | # mount –o iocharset=gb2312 codepage=936 /dev/hda5 /mnt/hda5  还有问题尝试  # mount –o iocharset=utf8 /dev/hda5 /mnt/hda5 |
| 普通用户不可写 | # mount –o umask=000, iocharset=cp936 /dev/hda5 /mnt/hda5 |
| 文件都变成短文件名 | 这是文件系统挂错的原因，将FAT32挂载成FAT16时就会出现这种情况，先umount，然后用 –t vfat 重新挂载即可解决问题。  CODE:  # mount –t vat /dev/hda5 /mnt/hda5 |
| 挂载U盘和mp3 | 如果计算机没有其它SCSI设备和usb外设的情况下，插入的U盘的设备路径是 /dev/sda1，用命令：  CODE:  # mkdir /mnt/u  # mount /dev/sda1 /mnt/u |
| 直接使用iso文件 | # mkdir /mnt/iso  # mount –o loop linux.iso /mnt/iso  当然，挂载以后挂载点/mnt/iso也是只读的。 |
| 挂载光驱和软驱 | # mkdir /mnt/cdrom  # mount /dev/hdc /mnt/cdrom //挂载光驱 //  # mkdir /mnt/floppy  # mount /dev/fd0 /mnt/floppy //挂载软驱 // |
| 开机自动挂载分区 | 将分区信息写到/etc/fstab文件中即可实现系统启动的自动挂载:  /dev/hda5 /mnt/hda5 vfat defaults,iocharset=cp936, rw 0 0 |
| 挂载samba 分区 | # mkdir /mnt/share  # mount -t smbfs -o username=root,password=abc,codepage=936,iocharset=gb2312//192.168.1.100/share /mnt/share |
| mount --bind是什么意思 | mount --bind 是将一个目录中的内容挂载到另一个目录上，用法是  CODE:  # mount --bind olddir newdir  这个命令使得自己搭建的FTP要共享某个目录的时候变得特别方便。如果要取消mount用命令：  CODE:  # mount --move olddir newdir 即可。  如果mount --bind 也想写入fstab中的话格式如下：  CODE:  olddir newdir none bind 0 0 |
| umount基本用法 | # umount /dev/hda5  # umount /mnt/hda5  # umount /dev/hda5 /mnt/hda5 |
| umount的时候老显示 device busy | 这是因为有程序正在访问这个设备，最简单的办法就是让访问该设备的程序退出以后再umount。可能有时候用户搞不清除究竟是什么程序在访问设备，如果用户不急着umount，则可以用:  CODE:  # umount -l /mnt/hda5  来卸载设备。选项 –l 并不是马上umount，而是在该目录空闲后再umount。还可以先用命令 ps aux 来查看占用设备的程序PID，然后用命令kill来杀死占用设备的进程，这样就umount的非常放心了。  方法一：  fuser -m -v /nfs\_old/  用户 进程号 权限 命令  /nfs\_old/: root 2699 ..c.. bash  root 2999 ..c.. su  如上所示，有两个进程占用了，将其kill掉，再重新取消挂载。  kill -9 2699 2999  umount /nfs\_old/  方法二：  umount -l /nfs\_old/ //等空闲再unmount  方法三：  fuser -km /nfs\_old/ //m表示mount，k表示kill |

VirtualBox和windows共享目录：

sudo mount -t vboxsf xyz /mnt/hgfs/share

xyz:windows共享目录名

/mnt/hgfs/share: linux上的路径

-t: type

## gdb

## grep

## apt

See <http://www.cnblogs.com/gbyukg/p/3248218.html>

默认下载的软件包路径：/var/cache/apt/archives

|  |  |
| --- | --- |
| 命令 | 说明 |
| apt-cache族 | 用于搜索哪些软件可以被安装在ubuntu或Debian上 |
| apt-cache pkgnames | 列出所有可用的软件包名 |
| apt-cache search vsftpd | 查找软件包vsftpd，并列出其信息 |
| apt-cache pkgnames vsftpd | 查找所有以vsftpd开头的软件包 |
| apt-cache show netcat | 查看软件包依赖关系，查看软件包信息 |
| apt-cahce showpkg vsftpd | showpkg子命令会检测指定软件包的依赖软件包是否已经被安装 |
| apt-cache stats | 查看软件包总体信息，stats子命令用于统计软件包总体信息 |
| apt-get族 | 获取相关软件包 |
| apt-get update | 更新系统软件包列表  update子命令会同步/etc/apt/sources.list文件中的软件包索引 |
| apt-get upgrade | 更新软件包  upgrade子命令会根据软件列表（update后的）更新当前系统中所有已安装的软件包，并同时更新和软件包所依赖的软件包 |
| apt-get install netcat | 安装或更新netcat软件包 |
| apt-get install packageName --no-upgrade | --no-upgrade子命令会阻止已经安装过的文件进行更新操作 |
| apt-get install packageName --only-upgrade | --only-upgrade子命令只会更新已经安装过的文件，并不会安装新文件 |
| apt-get install vsftpd=2.3.5... | 安装指定版本的包文件 |
| apt-get remove vsftpd | 移除软件包，但是保留相关配置文件信息 |
| apt-get purge vsftpd | 移除软件包和配置文件，或者使用：apt-get remove --purge vsftpd |
| apt-get clean | 删除所以已下载的软件包 |
| apt-get --download-only source vsftpd | 只下载软件源码包 |
| apt-get source vsftpd | 下载并解压包 |
| apt-get --compile source goaccess | 下载、解压并编译 |
| apt-get download nethogs | 仅将软件包下载到当前工作目录中 |
| apt-get changelog vsftpd | 查看软件包的日志信息 |
| apt-get check | 检查有没有broken的依赖 |
| apt-get build-dep netcat | 在当前系统中的本地包库中查看指定包的依赖包并对以来包进行安装 |

# Linux驱动开发

## Sample:

#include<linux/init.h>

#include<linux/module.h>

MODULE\_LICENSE("Dual BSD/GPL");

static int hello\_init(void)

{

printk(KERN\_ALERT "Hello, world\n");

return 0;

}

static void hello\_exit(void)

{

printk(KERN\_ALERT "Goodbye, cruel world\n");

}

module\_init(hello\_init);

module\_exit(hello\_exit);

## 编译

Makefile:

Obj-m := hello.o

#make -C ~/kernel-2.6 -M=’pwd’ modules

## 装载和卸载

#insmod hello.ko

#lsmod |grep hello

#rmmod hello

|  |  |  |
| --- | --- | --- |
| 工具 | 相同点 | 不同点 |
| insmod | 装载模块 | 只加载本模块 |
| modprobe | 考虑但钱模块的依赖，如果在该模块中存在内核中没有的symbol，则会在模块搜索路径中查找其他模块是否有该symbol，如果有则加载进来。  只能从已安装模块目录中搜索模块：/lib/modules/$(shell uname -r)/ |

## 模块层叠技术

一个模块在内核中导出符号，另一个层叠其上的模块使用其导出的符号，这种模块间相互依赖的技术。

相关宏：

|  |  |  |
| --- | --- | --- |
| EXPORT\_SYMBOL(name); | 导出符号到模块外部 | 只能在模块文件的全局部分导出，不能在函数内部导出 |
| EXPORT\_SYMBOL\_GPL(name); | 导出的符号只能被GPL许可证下的模块使用 |

## 查看printk的信息

#dmesg

## 用户空间编写驱动程序

通常，用户空间的驱动程序被实现为一个服务器进程，其任务是代替内核成为硬件控制的唯一代理。客户应用程序可连接到该服务器进程和设备执行实际的通讯，好的驱动程序可以允许对设备的并发访问。

## 驱动的调试

# 进程间通讯

| **名称** | **详细** |
| --- | --- |
| 信号 | 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。 |
|
| 管道 | 管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。 |
| 有名管道 | 有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。 |
| 消息队列 | 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。 |
| 共享内存 | 共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。 |
| 信号量 | 信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。 |
| socket | 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同主机间的进程通信。 |

## 信号

### 信号列表

/\* Signals. \*/

#define SIGHUP 1 /\* Hangup (POSIX). \*/

#define SIGINT 2 /\* Interrupt (ANSI). \*/

#define SIGQUIT 3 /\* Quit (POSIX). \*/

#define SIGILL 4 /\* Illegal instruction (ANSI). \*/

#define SIGTRAP 5 /\* Trace trap (POSIX). \*/

#define SIGABRT 6 /\* Abort (ANSI). \*/

#define SIGIOT 6 /\* IOT trap (4.2 BSD). \*/

#define SIGBUS 7 /\* BUS error (4.2 BSD). \*/

#define SIGFPE 8 /\* Floating-point exception (ANSI). \*/

#define SIGKILL 9 /\* Kill, unblockable (POSIX). \*/ 不可捕获

#define SIGUSR1 10 /\* User-defined signal 1 (POSIX). \*/

#define SIGSEGV 11 /\* Segmentation violation (ANSI). \*/

#define SIGUSR2 12 /\* User-defined signal 2 (POSIX). \*/

#define SIGPIPE 13 /\* Broken pipe (POSIX). \*/

#define SIGALRM 14 /\* Alarm clock (POSIX). \*/

#define SIGTERM 15 /\* Termination (ANSI). \*/

#define SIGSTKFLT 16 /\* Stack fault. \*/

#define SIGCLD SIGCHLD /\* Same as SIGCHLD (System V). \*/

#define SIGCHLD 17 /\* Child status has changed (POSIX). \*/

#define SIGCONT 18 /\* Continue (POSIX). \*/

#define SIGSTOP 19 /\* Stop, unblockable (POSIX). \*/ 不可捕获

#define SIGTSTP 20 /\* Keyboard stop (POSIX). \*/

#define SIGTTIN 21 /\* Background read from tty (POSIX). \*/

#define SIGTTOU 22 /\* Background write to tty (POSIX). \*/

#define SIGURG 23 /\* Urgent condition on socket (4.2 BSD). \*/

#define SIGXCPU 24 /\* CPU limit exceeded (4.2 BSD). \*/

#define SIGXFSZ 25 /\* File size limit exceeded (4.2 BSD). \*/

#define SIGVTALRM 26 /\* Virtual alarm clock (4.2 BSD). \*/

#define SIGPROF 27 /\* Profiling alarm clock (4.2 BSD). \*/

#define SIGWINCH 28 /\* Window size change (4.3 BSD, Sun). \*/

#define SIGPOLL SIGIO /\* Pollable event occurred (System V). \*/

#define SIGIO 29 /\* I/O now possible (4.2 BSD). \*/

#define SIGPWR 30 /\* Power failure restart (System V). \*/

#define SIGSYS 31 /\* Bad system call. \*/

#define SIGUNUSED 31

#define \_NSIG 65 /\* Biggest signal number + 1

(including real-time signals). \*/

#define SIGRTMIN (\_\_libc\_current\_sigrtmin ())

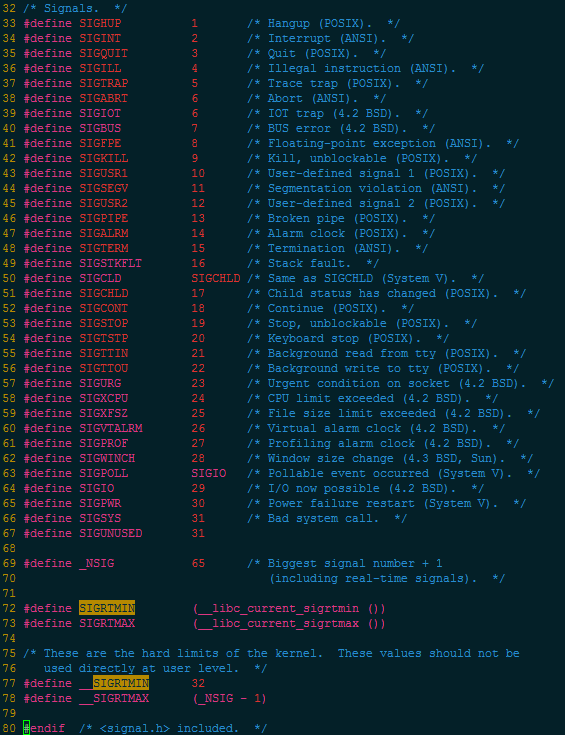
#define SIGRTMAX (\_\_libc\_current\_sigrtmax ())

/\* These are the hard limits of the kernel. These values should not be

used directly at user level. \*/

#define \_\_SIGRTMIN 32

#define \_\_SIGRTMAX (\_NSIG - 1)



See detail in file <bits/signum.h>

信号含义

(1) SIGHUP：当用户退出Shell时，由该Shell启发的所有进程都会接收到这个信号，默认动作为终止进程。

(2) SIGINT：用户按下组合键时，用户端时向正在运行中的由该终端启动的程序发出此信号。默认动作为终止进程。

(3) SIGQUIT：当用户按下组合键时产生该信号，用户终端向正在运行中的由该终端启动的程序发出此信号。默认动作为终止进程并产生core文件。

(4) SIGILL ：CPU检测到某进程执行了非法指令。默认动作为终止进程并产生core文件。

(5) SIGTRAP：该信号由断点指令或其他trap指令产生。默认动作为终止进程并产生core文件。

(6) SIGABRT：调用abort函数时产生该信号。默认动作为终止进程并产生core文件。

(7) SIGBUS：非法访问内存地址，包括内存地址对齐（alignment）出错，默认动作为终止进程并产生core文件。

(8) SIGFPE：在发生致命的算术错误时产生。不仅包括浮点运行错误，还包括溢出及除数为0等所有的算术错误。默认动作为终止进程并产生core文件。

(9) SIGKILL：无条件终止进程。本信号不能被忽略、处理和阻塞。默认动作为终止进程。它向系统管理员提供了一种可以杀死任何进程的方法。

(10) SIGUSR1：用户定义的信号，即程序可以在程序中定义并使用该信号。默认动作为终止进程。

(11) SIGSEGV：指示进程进行了无效的内存访问。默认动作为终止进程并使用该信号。默认动作为终止进程。

(12) SIGUSR2：这是另外一个用户定义信号，程序员可以在程序中定义并使用该信号。默认动作为终止进程。

(13) SIGPIPE：Broken pipe：向一个没有读端的管道写数据。默认动作为终止进程。

(14) SIGALRM：定时器超时，超时的时间由系统调用alarm设置。默认动作为终止进程。

(15) SIGTERM：程序结束(terminate)信号，与SIGKILL不同的是，该信号可以被阻塞和处理。通常用来要求程序正常退出。执行Shell命令kill时，缺少产生这个信号。默认动作为终止进程。

(16) SIGCHLD：子程序结束时，父进程会收到这个信号。默认动作为忽略该信号。

(17) SIGCONT：让一个暂停的进程继续执行。

(18) SIGSTOP：停止(stopped)进程的执行。注意它和SIGTERM以及SIGINT的区别：该进程还未结束，只是暂停执行。本信号不能被忽略、处理和阻塞。默认作为暂停进程。

(19) SIGTSTP：停止进程的动作，但该信号可以被处理和忽略。按下组合键时发出该信号。默认动作为暂停进程。

(20) SIGTTIN：当后台进程要从用户终端读数据时，该终端中的所有进程会收到SIGTTIN信号。默认动作为暂停进程。

(21) SIGTTOU：该信号类似于SIGTIN，在后台进程要向终端输出数据时产生。默认动作为暂停进程。

(22) SIGURG：套接字（socket）上有紧急数据时，向当前正在运行的进程发出此信号，报告有紧急数据到达。默认动作为忽略该信号。

(23) SIGXCPU：进程执行时间超过了分配给该进程的CPU时间，系统产生该信号并发送给该进程。默认动作为终止进程。

(24) SIGXFSZ：超过文件最大长度的限制。默认动作为终止进程并产生core文件。

(25) SIGVTALRM：虚拟时钟超时时产生该信号。类似于SIGALRM，但是它只计算该进程占有用的CPU时间。默认动作为终止进程。

(26) SIGPROF：类似于SIGVTALRM，它不仅包括该进程占用的CPU时间还抱括执行系统调用的时间。默认动作为终止进程。

(27) SIGWINCH：窗口大小改变时发出。默认动作为忽略该信号。

(28) SIGIO：此信号向进程指示发出一个异步IO事件。默认动作为忽略。

(29) SIGPWR：关机。默认动作为终止进程。

(30) SIGRTMIN~SIGRTMAX：Linux的实时信号，它没有固定的含义(或者说可以由用户自由使用)。

注意，Linux线程机制使用了前3个实时信号。所有的实时信号的默认动作都是终止进程。

### 信号分类

|  |  |
| --- | --- |
| **可靠信号（非实时）** | 会丢失的，不可排队的；继承自UNIX，小于SIGRTMIN(32)的信号  非实时信号都不支持排队，都是不可靠信号； |
| **不可靠信号（实时）** | 不会丢失的，可排队的；Linux自定义的，SIGRTMIN(32)~SIGRTMAX(64)之间的信号  实时信号都支持排队，都是可靠信号。 |

### 信号优先级

信号实质上是软中断，中断有优先级，信号也有优先级。如果一个进程有多个未决信号（pending），则对于同一个未决的实时信号，内核将按照发送的顺序来递送信号。如果存在多个未决信号，则值（或者说编号）越小的越先被递送。如果即存在不可靠信号，又存在可靠信号（实时信号），虽然POSIX对这一情况没有明确规定，但Linux系统和大多数遵循POSIX标准的操作系统一样，将优先递送不可靠信号。

### 信号的处理流程

* 信号诞生

信号事件的发生有两个来源：

**硬件来源**(比如我们按下了键盘或者其它硬件故障)；

**软件来源**，最常用发送信号的系统函数是kill, raise, alarm和setitimer以及sigqueue函数，软件来源还包括一些非法运算等操作。

这里按发出信号的**原因**简单分类，以了解各种信号：

（1） 与进程终止相关的信号。当进程退出，或者子进程终止时，发出这类信号。

（2） 与进程例外事件相关的信号。如进程越界，或企图写一个只读的内存区域（如程序正文区），或执行一个特权指令及其他各种硬件错误。

（3） 与在系统调用期间遇到不可恢复条件相关的信号。如执行系统调用exec时，原有资源已经释放，而目前系统资源又已经耗尽。

（4） 与执行系统调用时遇到非预测错误条件相关的信号。如执行一个并不存在的系统调用。

（5） 在用户态下的进程发出的信号。如进程调用系统调用kill向其他进程发送信号。

（6） 与终端交互相关的信号。如用户关闭一个终端，或按下break键等情况。

（7） 跟踪进程执行的信号。

* 信号在进程中注册

在进程表的表项中有一个软中断信号域，该域中每一位对应一个信号。内核给一个进程发送软中断信号的方法，是在进程所在的进程表项的信号域设置对应于该信号的位。如果信号发送给一个正在睡眠的进程，如果进程睡眠在可被中断的优先级上，则唤醒进程；否则仅设置进程表中信号域相应的位，而不唤醒进程。如果发送给一个处于可运行状态的进程，则只置相应的域即可。

进程的task\_struct结构中有关于本进程中未决信号的数据成员： struct sigpending pending：

struct sigpending{

struct sigqueue \*head, \*tail;

sigset\_t signal;

};

第三个成员是进程中所有未决信号集，第一、第二个成员分别指向一个sigqueue类型的结构链（称之为"未决信号信息链"）的首尾，信息链中的每个sigqueue结构刻画一个特定信号所携带的信息，并指向下一个sigqueue结构:

struct sigqueue{

struct sigqueue \*next;

siginfo\_t info;

}

 /\* A `sigset\_t' has a bit for each signal.  \*/  
 27   
 28 # define \_SIGSET\_NWORDS (1024 / (8 \* sizeof (unsigned long int)))  
 29 typedef struct  
 30   {  
 31     unsigned long int \_\_val[\_SIGSET\_NWORDS];  
 32   } \_\_sigset\_t;  
 33   
 34 #endif

信号在进程中注册指的就是信号值加入到进程的未决信号集sigset\_t signal（每个信号占用一位）中，并且信号所携带的信息被保留到未决信号信息链的某个sigqueue结构中。只要信号在进程的未决信号集中，表明进程已经知道这些信号的存在，但还没来得及处理，或者该信号被进程阻塞。

|  |  |  |
| --- | --- | --- |
| 信号类型 | 如果已经注册过 | 多个相同信号占用sigqueue结构个数 |
| 实时信号 | 再次注册 | 占用多个 |
| 非实时信号 | 丢弃该信号 | 最多一个 |

总之信号注册与否，与发送信号的函数（如kill()或sigqueue()等）以及信号安装函数（signal()及sigaction()）无关，只与信号值有关（信号值小于SIGRTMIN的信号最多只注册一次，信号值在SIGRTMIN及SIGRTMAX之间的信号，只要被进程接收到就被注册）

* 信号的执行和注销

| **对象** | **详细** |
| --- | --- |
| **进程状态** | 内核处理一个进程收到的软中断信号是在该进程的上下文中，因此，进程必须处于运行状态。 |
| **处理时机** | 当其用于被信号唤醒或者正常调度重新获得CPU时，在其从内核空间返回到用户空间时会检测是否有信号等待处理。当一个进程在内核态下运行时，软中断信号并不立即起作用，要等到将返回用户态时才处理。进程只有处理完信号才会返回用户态，进程在用户态下不会有未处理完的信号。 |
| **注销非实时信号** | 由于在未决信号信息链中最多只占用一个sigqueue结构，因此该结构被释放后，应该把信号在进程未决信号集中删除（信号注销完毕）； |
| **注销实时信号** | 如果只占用一个sigqueue结构（进程只收到该信号一次），则执行完相应的处理函数后应该把信号在进程的未决信号集中删除（信号注销完毕）；  否则待该信号的所有sigqueue处理完毕后再在进程的未决信号集中删除该信号。 |
| **屏蔽的信号** | 当所有未被屏蔽的信号都处理完毕后，即可返回用户空间。对于被屏蔽的信号，当取消屏蔽后，在返回到用户空间时会再次执行上述检查处理的一套流程。 |
| **内核处理信号的时机** | 内核处理一个进程收到的信号的时机是在一个进程从内核态返回用户态时。 |
| **处理信号三种类型** | 进程接收到信号后退出； |
| 进程忽略该信号；当进程接收到一个它忽略的信号时，进程丢弃该信号，就象没有收到该信号似的继续运行。 |
| 进程收到信号后执行用户函数（用系统调用signal设定）。  如果进程收到一个要捕捉的信号，那么进程从内核态返回用户态时执行用户定义的函数。  而且执行用户定义的函数的方法很巧妙，内核是在用户栈上创建一个新的层，该层中将返回地址的值设置成用户定义的处理函数的地址，这样进程从内核返回弹出栈顶时就返回到用户定义的函数处，从函数返回再弹出栈顶时，才返回原先进入内核的地方。  这样做的原因是用户定义的处理函数不能且不允许在内核态下执行（如果用户定义的函数在内核态下运行的话，用户就可以获得任何权限）。 |

### 系统函数

#### 信号安装

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <signal.h> | |
| **函数原型** | typedef void (\*sighandler\_t)(int);  sighandler\_t signal(int signum, sighandler\_t handler); | |
| **参数** | signum | 信号值 |
| handler | 处理函数  可以忽略该信号（参数设为SIG\_IGN）；可以采用系统默认方式处理信号(参数设为SIG\_DFL)； |
| **返回值** | returns the previous value of the signal handler, or SIG\_ERR on error | |

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <signal.h> | |
| **函数原型** | int sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact); | |
| **参数** | signum | signal value |
| act | pointer to new struct sigaction |
| oldact | pointer to previous struct sigaction |
| **返回值** | returns 0 on success and -1 on error. | |

struct sigaction {

void (\*sa\_handler)(int);

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);/\*第一个是信号值，第三个参数没有使用\*/

sigset\_t sa\_mask;

int sa\_flags;

void (\*sa\_restorer)(void);

};

siginfo\_t {

int si\_signo; /\* 信号值，对所有信号有意义\*/

int si\_errno; /\* errno值，对所有信号有意义\*/

int si\_code; /\* 信号产生的原因，对所有信号有意义\*/

pid\_t si\_pid; /\* 发送信号的进程ID,对kill(2),实时信号以及SIGCHLD有意义 \*/

uid\_t si\_uid; /\* 发送信号进程的真实用户ID，对kill(2),实时信号以及SIGCHLD有意义 \*/

int si\_status; /\* 退出状态，对SIGCHLD有意义\*/

clock\_t si\_utime; /\* 用户消耗的时间，对SIGCHLD有意义 \*/

clock\_t si\_stime; /\* 内核消耗的时间，对SIGCHLD有意义 \*/

sigval\_t si\_value; /\* 信号值，对所有实时有意义，是一个联合数据结构，

/\* 可以为一个整数（由si\_int标示，也可以为一个指针，由si\_ptr标示）\*/

void \* si\_addr; /\* 触发fault的内存地址，对SIGILL,SIGFPE,SIGSEGV,SIGBUS 信号有意义\*/

int si\_band; /\* 对SIGPOLL信号有意义 \*/

int si\_fd; /\* 对SIGPOLL信号有意义 \*/

}

#### 信号发送

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/types.h>  #include <signal.h> | |
| **函数原型** | int kill(pid\_t pid,int signo) | |
| **参数** | pid | pid>0 进程ID为pid的进程  pid=0 同一个进程组的进程  pid<0 pid!=-1 进程组ID为 -pid的所有进程  pid=-1 除发送进程自身外，所有进程ID大于1的进程 |
| signo | signal value  Sinno是信号值，当为0时（即空信号），实际不发送任何信号，但照常进行错误检查，因此，可用于检查目标进程是否存在，以及当前进程是否具有向目标发送信号的权限（root权限的进程可以向任何进程发送信号，非root权限的进程只能向属于同一个session或者同一个用户的进程发送信号）。 |
| **返回值** | On success (at least one signal was sent), zero is returned.  On error, -1 is returned, and errno is set appropriately. | |
| **errno** | EINVAL：指定的信号sig无效。  ESRCH：参数pid指定的进程或进程组不存在。注意，在进程表项中存在的进程，可能是一个还没有被wait收回，但已经终止执行的僵死进程。  EPERM： 进程没有权力将这个信号发送到指定接收信号的进程。因为，一个进程被允许将信号发送到进程pid时，必须拥有root权力，或者是发出调用的进程的UID 或EUID与指定接收的进程的UID或保存用户ID（savedset-user-ID）相同。如果参数pid小于-1，即该信号发送给一个组，则该错误表示组中有成员进程不能接收该信号。 | |

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/types.h>  #include <signal.h> | |
| **函数原型** | int sigqueue(pid\_t pid, int sig, const union sigval val) | |
| **参数** | pid | process id |
| sig | signal value |
| val | The value argument is used to specify an accompanying item of data (either an integer or a pointer value) to be sent with the signal, and has the following type: |
| **返回值** | On success, sigqueue() returns 0, indicating that the signal was successfully queued to the receiving process.  Otherwise -1 is returned and errno is set to indicate the error. | |

说明：sigqueue()与函数sigaction()配合使用

union sigval {

int sival\_int;

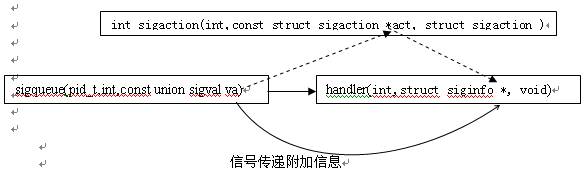
void \*sival\_ptr;

};

Note:

sigqueue()比kill()传递了更多的附加信息，但sigqueue()只能向一个进程发送信号，而不能发送信号给一个进程组。

在调用sigqueue时，sigval\_t指定的信息会拷贝到对应的sigaction()指定的信号处理函数的siginfo\_t结构中，这样信号处理函数就可以处理这些信息了。由于sigqueue系统调用支持发送带参数信号，所以比kill()系统调用的功能要灵活和强大得多。



|  |  |  |
| --- | --- | --- |
| **头文件** | #include <unistd.h> | |
| **函数原型** | unsigned int alarm(unsigned int seconds) | |
| **参数** | seconds | 让system在若干秒后发给调用进程一个SIGALRM信号 |
| **返回值** | 如果指定的参数seconds为0，则不再发送 SIGALRM信号。后一次设定将取消前一次的设定。该调用返回值为上次定时调用到发送之间剩余的时间，或者因为没有前一次定时调用而返回0。 | |
| **注意** | 在使用时，alarm只设定为发送一次信号，如果要多次发送，就要多次使用alarm调用。 | |

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/time.h> | |
| **函数原型** | int setitimer(int which, const struct itimerval \*value, struct itimerval \*ovalue); | |
| **参数** | which | IMER\_REAL：按实际时间计时，计时到达将给进程发送SIGALRM信号。 |
| ITIMER\_VIRTUAL：仅当进程执行时才进行计时。计时到达将发送SIGVTALRM信号给进程。 |
| ITIMER\_PROF：当进程执行时和系统为该进程执行动作时都计时。与ITIMER\_VIRTUAL是一对，该定时器经常用来统计进程在用户态和内核态花费的时间。计时到达将发送SIGPROF信号给进程。 |
| value | value用来指明定时器的时间 |
| ovalue | 参数ovalue如果不为空，则其中保留的是上次调用设定的值。  Timers decrement from it\_value to zero, generate a signal, and reset to it\_interval.  A timer which is set to zero (it\_value is zero or the timer expires and it\_interval is zero) stops. |
| **返回值** | On success, zero is returned. On error, -1 is returned, and errno is set appropriately. | |
| **errno** | EFAULT：参数value或ovalue是无效的指针。  EINVAL：参数which不是ITIMER\_REAL、ITIMER\_VIRT或ITIMER\_PROF中的一个。 | |

struct itimerval {

struct timeval it\_interval; /\* next value \*/

struct timeval it\_value; /\* current value \*/

};

struct timeval {

long tv\_sec; /\* seconds \*/

long tv\_usec; /\* microseconds \*/

};

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <stdlib.h> | |
| **函数原型** | void abort(void); | |
| **参数** | void |  |
| **返回值** | void | |
| **note** | 向进程发送SIGABORT信号，默认情况下进程会异常退出，当然可定义自己的信号处理函数。  即使SIGABORT被进程设置为阻塞信号，调用abort()后，SIGABORT仍然能被进程接收(The abort() first unblocks the SIGABRT signal, and then raises that signal for the calling process.)。  即使自定义的信号处理函数捕获该信号，一旦返回，该进程依然会退出。因为，It does this by restoring the default disposition for SIGABRT and then raising the signal for a second time.  反正，无论怎样该进程都会被终止，且SIGABRT会被发出来。 | |

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <signal.h> | |
| **函数原型** | int raise(int sig); | |
| **参数** | sig | sig value |
| **返回值** | returns 0 on success, and non-zero for failure | |
| **说明** | The raise() function sends a signal to the calling process or thread. In a single-threaded program it is equivalent to  kill(getpid(), sig);  In a multithreaded program it is equivalent to  pthread\_kill(pthread\_self(), sig);  If the signal causes a handler to be called, raise() will only return after the signal handler has returned. | |

#### 信号集操作

|  |  |
| --- | --- |
| **API** | **说明** |
| int sigemptyset(sigset\_t \*set)； | 初始化由set指定的信号集，信号集里面的所有信号被清空； |
| int sigfillset(sigset\_t \*set)； | 调用该函数后，set指向的信号集中将包含linux支持的64种信号； |
| int sigaddset(sigset\_t \*set, int signum) | 在set指向的信号集中加入signum信号； |
| int sigdelset(sigset\_t \*set, int signum) | 在set指向的信号集中删除signum信号； |
| int sigismember(const sigset\_t \*set, int signum)； | 判定信号signum是否在set指向的信号集中。 |

头文件#include <signal.h>

#### 信号阻塞与信号未决

|  |  |
| --- | --- |
| **API** | **说明** |
| int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset)); | SIG\_BLOCK 在进程当前阻塞信号集中添加set指向信号集中的信号  SIG\_UNBLOCK 如果进程阻塞信号集中包含set指向信号集中的信号，则解除对该信号的阻塞  SIG\_SETMASK 更新进程阻塞信号集为set指向的信号集 |
| int sigpending(sigset\_t \*set)); | 获得当前已递送到进程，却被阻塞的所有信号，在set指向的信号集中返回结果 |
| int sigsuspend(const sigset\_t \*mask)); | 用于在接收到某个信号之前, 临时用mask替换进程的信号掩码, 并暂停进程执行，直到收到信号为止。sigsuspend 返回后将恢复调用之前的信号掩码。信号处理函数完成后，进程将继续执行。该系统调用始终返回-1，并将errno设置为EINTR |

## 管道

1.只能用于具有亲缘关系的进程之间的通信

2.半双工通信模式

3.一种特殊的文件，是一种只存在于内核中的读写函数

### 创建管道

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <unistd.h> | |
| **函数原型** | int pipe(int filedes[2]); | |
| **参数** | filedes[0] | for reading |
| filedes[1] | for writing |
| **返回值** | On success, zero is returned. On error, -1 is returned, and errno is set appropriately. | |

### 读管道

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <unistd.h> | |
| **函数原型** | ssize\_t read(int fd, void \*buf, size\_t count); | |
| **参数** | fd | file descriptor |
| buf | pointer to buffer |
| count | If count is zero, read() returns zero and has no other results.  If count is greater than SSIZE\_MAX, the result is unspecified. |
| **返回值** | On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.  On error, -1 is returned, and errno is set appropriately. | |

### 写管道

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <unistd.h> | |
| **函数原型** | ssize\_t write(int fd, const void \*buf, size\_t count); | |
| **参数** | fd | file descriptor |
| buf | pointer to buffer |
| count | writes up to count bytes to the file referenced by the file descriptor fd from the buffer starting at buf. |
| **返回值** | On success, the number of bytes written are returned (zero indicates nothing was written). On error, -1 is returned, and errno is set appropriately. If count is zero and the file descriptor refers to a regular file, 0 may be returned, or an error could be detected. For a special file, the results are not portable. | |

## 有名管道

1. 使不相关的两个进程彼此通信：

a. 通过路径名指出，在文件系统中可见

b. 管道建立后，两进程可按普通文件一样对其操作

2. FIFO遵循先进先出规则：

a. 对管道读从开始处返回数据

b. 对管道写则把数据添加到末尾

c. 不支持如lseek()等文件定位操作

### 创建管道

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/types.h>  #include <sys/stat.h> | |
| **函数原型** | int mkfifo(const char \*pathname, mode\_t mode); | |
| **参数** | pathname | mkfifo() makes a FIFO special file with name pathname |
| mode | mode specifies the FIFO's permissions. |
| **返回值** | On success mkfifo() returns 0. In the case of an error, -1 is returned (in which case, errno is set appropriately). | |
| **说明** | any process can open it for reading or writing  Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa. | |

### 打开管道

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h> | |
| **函数原型** | int open(const char \*pathname, int flags);  int open(const char \*pathname, int flags, mode\_t mode); | |
| **参数** | pathname | path name |
| flags | must include one of the following access modes: O\_RDONLY, O\_WRONLY, or O\_RDWR. |
| mode | specifies the permissions to use in case a new file is created. |
| **返回值** | return the new file descriptor, or -1 if an error occurred (in which case, errno is set appropriately) | |

### 写管道

See above

### 读管道

See above

### 删除管道

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <stdio.h> | |
| **函数原型** | int remove(const char \*path); | |
| **参数** | path | If path does not name a directory, remove(path) shall be equivalent to unlink(path).  If path names a directory, remove(path) shall be equivalent to rmdir(path). |
| **返回值** | On success, zero is returned. On error, -1 is returned, and errno is set appropriately | |

### Broken pipe管道断裂

只有在管道的读端存在时，向管道中写入数据才有意义。否则，向管道中写入数据的进程将收到内核传来的SIFPIPE信号，应用程序可以处理该信号，也可以忽略（默认动作则是应用程序终止）。

常见的几种：

1）broken pipe的字面意思是“管道破裂”。broken pip的原因是该管道的读端被关闭。

2）broken pipe经常发生socket关闭之后（或者其他的描述符关闭之后）的write操作中。

3）发生broken pipe错误时，进程收到SIGPIPE信号，默认动作是进程终止。

4）broken pipe最直接的意思是：写入端出现的时候，另一端却休息或退出了，因此造成没有及时取走管道中的数据，从而系统异常退出；

### 比较

|  |  |
| --- | --- |
| **普通文件** | 不会出现阻塞 |
| **管道文件** | 读写中可能有阻塞（非阻塞标志：在open()函数中设定为O\_NONBLOCK） |

|  |  |  |
| --- | --- | --- |
|  | **阻塞打开** | **非阻塞打开** |
| **读进程** | 当前FIFO内没有数据，则对读进程而言将一直阻塞到有数据写入 | 不论FIFO内是否有数据，读进程都会立即执行读操作。即如果FIFO内没有数据，则读函数将立刻返回0 |
| **写进程** | 写操作将一直阻塞到数据可以被写入 | 若该管道是非阻塞打开而不能写入全部数据，则读操作进行部分写入或者调用失败 |

## 消息队列

### 消息队列的创建

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h> | |
| **函数原型** | int msgget(key\_t key, int msgflg); | |
| **参数** | key | The msgget() function shall return the message queue identifier associated with the argument key. |
| msgflg | IPC\_CREAT and IPC\_EXCL and a message queue already exists for key, then msgget() fails with errno set to EEXIST |
| **返回值** | If successful, the return value will be the message queue identifier (a non-negative integer), otherwise -1 with errno indicating the error. | |

### 消息发送

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h> | |
| **函数原型** | int msgsnd(int msqid, const void \*msgp, size\_t msgsz, int msgflg); | |
| **参数** | msqid | msg queue id |
| msgp | pointer to msg whose stucture should be same as below  (points to a user-defined buffer that contains first a field of type long specifying the type of the message, and then a data portion that holds the data bytes of the message) |
| msgsz | msg size |
| msgflg | msg flag |
| **返回值** | Upon successful completion, msgsnd() shall return 0; otherwise, no message shall be sent, msgsnd() shall return -1, and errno shall be set to indicate the error. | |
| **说明** | The structure member mtype is a non-zero positive type long that can be used by the receiving process for message selection. | |

struct mymsg {

long mtype; /\* Message type. \*/

char mtext[1]; /\* Message text. \*/

}

### 消息接收

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h> | |
| **函数原型** | ssize\_t msgrcv(int msqid, void \*msgp, size\_t msgsz, long msgtyp, int msgflg); | |
| **参数** | msqid | msg queue id |
| msgp | pointer to the msg |
| msgsz | size of msg |
| msgtyp | 0, then the first message in the queue is read. |
| greater than 0, then the first message in the queue of type msgtyp is read, unless MSG\_EXCEPT was specified in msgflg, in which case the first message in the queue of type not equal to msgtyp will be read. |
| less than 0, then the first message in the queue with the lowest type less than or equal to the absolute value of msgtyp will be read. |
| msgflg | The msgflg argument is a bit mask constructed by ORing together zero or more of the following flags: |
| IPC\_NOWAIT  Return immediately if no message of the requested type is in the queue. The system call fails with errno set to ENOMSG. |
| MSG\_EXCEPT  Used with msgtyp greater than 0 to read the first message in the queue with message type that differs from msgtyp. |
| MSG\_NOERROR  To truncate the message text if longer than msgsz bytes. |
| **返回值** | Upon successful completion, msgrcv() shall return a value equal to the number of bytes actually placed into the buffer mtext.  Otherwise, no message shall be received, msgrcv() shall return (ssize\_t)-1, and errno shall be set to indicate the error. | |

### 删除消息队列

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/msg.h> | |
| **函数原型** | int msgctl(int msqid, int cmd, struct msqid\_ds \*buf); | |
| **参数** | msqid | queue id |
| cmd  IPC\_STAT | 将kernel中和msqid\_ds对应的的信息拷贝到buf中，要求caller必须对queue有read权限 |
| IPC\_SET | 将buf中的数据写入kernel中，一下这些members将会被更新：  msg\_qbytes, msg\_perm.uid, msg\_perm.gid, and (the least significant 9 bits of) msg\_perm.mode.  要求caller的UID需要匹配queue的owner(msg\_perm.uid)或creator (msg\_perm.cuid)；或是获得特权(privileged); |
| IPC\_RMID | Immediately remove the message queue, awakening all waiting reader and writer processes |
| IPC\_INFO (Linux-specific) | 将system-wide queue limits and parameters返回到buf中 |
| MSG\_INFO (Linux-specific) | Returns a msginfo structure containing the same information as for IPC\_INFO, except that the following fields are returned with information about system resources consumed by message queues:  the msgpool field returns the number of message queues that currently exist on the system;  the msgmap field returns the total number of messages in all queues on the system;  and the msgtql field returns the total number of bytes in all messages in all queues on the system. |
| MSG\_STAT (Linux-specific) | Returns a msqid\_ds structure as for IPC\_STAT. However, the msqid argument is not a queue identifier, but instead an index into the kernel?. internal array that maintains information about all message queues on the system. |
| buf | pointer to the following structure |
| **返回值** | IPC\_STAT, IPC\_SET, and IPC\_RMID | On success, IPC\_STAT, IPC\_SET, and IPC\_RMID return 0. |
| IPC\_INFO or MSG\_INFO | A successful IPC\_INFO or MSG\_INFO operation returns the index of the highest used entry in the kernel's internal array recording information about all message queues. |
| MSG\_STAT | A successful MSG\_STAT operation returns the identifier of the queue whose index was given in msqid. |
| On error | On error, -1 is returned with errno indicating the error. |

The msqid\_ds data structure is defined in <sys/msg.h> as follows:

struct msqid\_ds {

struct ipc\_perm msg\_perm; /\* Ownership and permissions \*/

time\_t msg\_stime; /\* Time of last msgsnd(2) \*/

time\_t msg\_rtime; /\* Time of last msgrcv(2) \*/

time\_t msg\_ctime; /\* Time of last change \*/

unsigned long \_\_msg\_cbytes; /\* Current number of bytes in

queue (non-standard) \*/

msgqnum\_t msg\_qnum; /\* Current number of messages

in queue \*/

msglen\_t msg\_qbytes; /\* Maximum number of bytes

allowed in queue \*/

pid\_t msg\_lspid; /\* PID of last msgsnd(2) \*/

pid\_t msg\_lrpid; /\* PID of last msgrcv(2) \*/

};

The ipc\_perm structure is defined in <sys/ipc.h> as follows (the highlighted in green fields are settable using IPC\_SET):

struct ipc\_perm {

key\_t \_\_key; /\* Key supplied to msgget(2) \*/

uid\_t uid; /\* Effective UID of owner \*/

gid\_t gid; /\* Effective GID of owner \*/

uid\_t cuid; /\* Effective UID of creator \*/

gid\_t cgid; /\* Effective GID of creator \*/

unsigned short mode; /\* Permissions \*/

unsigned short \_\_seq; /\* Sequence number \*/

};

This structure is of type msginfo (thus, a cast is required), defined in <sys/msg.h> if the \_GNU\_SOURCE feature test macro is defined:

struct msginfo {

int msgpool; /\* Size in kibibytes of buffer pool

used to hold message data;

unused within kernel \*/

int msgmap; /\* Maximum number of entries in message

map; unused within kernel \*/

int msgmax; /\* Maximum number of bytes that can be

written in a single message \*/

int msgmnb; /\* Maximum number of bytes that can be

written to queue; used to initialize

msg\_qbytes during queue creation

(msgget(2)) \*/

int msgmni; /\* Maximum number of message queues \*/

int msgssz; /\* Message segment size;

unused within kernel \*/

int msgtql; /\* Maximum number of messages on all queues

in system; unused within kernel \*/

unsigned short int msgseg;

/\* Maximum number of segments;

unused within kernel \*/

};

### 和有名管道的比较

消息队列跟命名管道有不少的**相同之处**

1、通过与命名管道一样，消息队列进行通信的进程可以是不相关的进程，

2、同时它们都是通过发送和接收的方式来传递数据的。在命名管道中，发送数据用write，接收数据用read，则在消息队列中，发送数据用msgsnd，接收数据用msgrcv。

3、而且它们对每个数据都有一个最大长度的限制。

与命名管道相比，消息队列的**优势**在于

1、消息队列也可以独立于发送和接收进程而存在，从而消除了在同步命名管道的打开和关闭时可能产生的困难。

2、同时通过发送消息还可以避免命名管道的同步和阻塞问题，不需要由进程自己来提供同步方法。

3、接收程序可以通过消息类型有选择地接收数据，而不是像命名管道中那样，只能默认地接收。

## 共享内存

### 创建共享内存

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/ipc.h>  #include <sys/shm.h> | |
| **函数原型** | int shmget(key\_t key, size\_t size, int shmflg); | |
| **参数** | key | key |
| size | shm size will be equal to the value of size rounded up to a multiple of PAGE\_SIZE |
| shmflg | IPC\_CREAT:创建shm  IPC\_EXCL:如果设置了该flag，且shm已经存在，仍然要create，则报错 |
| **返回值** | Upon successful completion, shmget() shall return a non-negative integer, namely a shared memory identifier; otherwise, it shall return -1 and set errno to indicate the error. | |

### 附着共享内存

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/ipc.h>  #include <sys/shm.h> | |
| **函数原型** | void \*shmat(int shmid, const void \*shmaddr, int shmflg); | |
| **参数** | shmid | id |
| shmaddr | If shmaddr is NULL, the system chooses a suitable (unused) address at which to attach the segment.  If shmaddr isn't NULL and SHM\_RND is specified in shmflg, the attach occurs at the address equal to shmaddr rounded down to the nearest multiple of SHMLBA. |
| shmflg | shmflg &SHM\_RND :  非0，attach地址(shmaddr -((uintptr\_t)shmaddr %SHMLBA))  为0，the segment is attached at the address given by shmaddr  SHM\_RDONLY：  非0，the segment is attached for reading  为0，the segment is attached for reading and writing |
| **返回值** | On success shmat() returns the address of the attached shared memory segment; on error (void \*) -1 is returned, and errno is set to indicate the cause of the error. | |

### 去附着共享内存

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/ipc.h>  #include <sys/shm.h> | |
| **函数原型** | int shmdt(const void \*shmaddr); | |
| **参数** | shmaddr | shm address |
| **返回值** | Upon successful completion, shmdt() shall decrement the value of shm\_nattch in the data structure associated with the shared memory ID of the attached shared memory segment and return 0. | |
| Otherwise, the shared memory segment shall not be detached, shmdt() shall return -1, and errno shall be set to indicate the error. | |

### 删除共享内存

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/ipc.h>  #include <sys/shm.h> | |
| **函数原型** | int shmctl(int shmid, int cmd, struct shmid\_ds \*buf); | |
| **参数** | shmid | shm id |
| cmd | IPC\_STAT:读取shm的状态信息  IPC\_SET:设置相关信息  IPC\_RMID:删除shm |
| buf | pointer to shmid\_ds structure, see below |
| **返回值** | Upon successful completion, shmctl() shall return 0; otherwise, it shall return -1 and set errno to indicate the error. | |

The buf argument is a pointer to a shmid\_ds structure, defined in <sys/shm.h> as follows:

struct shmid\_ds {

struct ipc\_perm shm\_perm; /\* Ownership and permissions \*/

size\_t shm\_segsz; /\* Size of segment (bytes) \*/

time\_t shm\_atime; /\* Last attach time \*/

time\_t shm\_dtime; /\* Last detach time \*/

time\_t shm\_ctime; /\* Last change time \*/

pid\_t shm\_cpid; /\* PID of creator \*/

pid\_t shm\_lpid; /\* PID of last shmat()/shmdt() \*/

shmatt\_t shm\_nattch; /\* No. of current attaches \*/

...

};

The ipc\_perm structure is defined in <sys/ipc.h> as follows (the highlighted fields are settable using IPC\_SET):

struct ipc\_perm {

key\_t key; /\* Key supplied to shmget() \*/

uid\_t uid; /\* Effective UID of owner \*/

gid\_t gid; /\* Effective GID of owner \*/

uid\_t cuid; /\* Effective UID of creator \*/

gid\_t cgid; /\* Effective GID of creator \*/

unsigned short mode; /\* Permissions + SHM\_DEST and

SHM\_LOCKED flags \*/

unsigned short seq; /\* Sequence number \*/

};

## 信号量

### 分类

Linux提供两种信号量

|  |  |  |
| --- | --- | --- |
| 内核信号量（内核态） | 由内核控制路径使用 | |
| 用户信号量（用户态） | POSIX信号量 | 有名信号量 |
| 无名信号量 |
| SYSTEM V信号量 | |

有名信号量：其值保存在文件中, 所以它可以用于线程也可以用于进程间的同步。

无名信号量：其值保存在内存中。

### 内核信号量

当内核控制路径试图获取内核信号量锁保护的忙资源时，相应的进程就被挂起。只有在资源被释放时，进程才再次变为可运行。

* 只有可以睡眠的函数才能获取内核信号量；
* 中断处理程序和可延迟函数都不能使用内核信号量。

不可能在内核之外使用

**信号量结构**

<asm/semaphore.h>

struct semaphore {  
　　 atomic\_t count; //相当于信号量的值，大于0，资源空闲；

//等于0，资源忙，但没有进程等待这个保护的资源；

//小于0，资源不可用，并至少有一个进程等待资源。  
　　 int sleepers; //存放一个标志，表示是否有一些进程在信号量上睡眠  
　　 wait\_queue\_head\_t wait;//存放等待队列链表的地址，当前等待资源的所有睡眠进程都会放在这个链表中  
}

上面已经提到了内核信号量使用了等待队列wait\_queue来实现阻塞操作。当某任务由于没有某种条件没有得到满足时，它就被挂到等待队列中睡眠。当条件得到满足时，该任务就被移出等待队列，此时并不意味着该任务就被马上执行，因为它又被移进工作队列中等待CPU资源，在适当的时机被调度。 内核信号量是在内部使用等待队列的，也就是说该等待队列对用户是隐藏的，无须用户干涉。

**相关函数**

|  |  |
| --- | --- |
| 初始化 | void sema\_init(struct semaphore \*sem, int val);//val is the value to init the semaphore |
| void init\_MUTEX (struct semaphore \*sem); //将sem的值置为1，表示资源空闲 |
| void init\_MUTEX\_LOCKED (struct semaphore \*sem); //将sem的值置为0，表示资源忙 |
| 申请信号量 | void down(struct semaphore \* sem); // 可引起睡眠 |
| int down\_interruptible(struct semaphore \* sem); // down\_interruptible能被信号打断 |
| int down\_trylock(struct semaphore \* sem); // 非阻塞函数，不会睡眠。无法锁定资源则马上返回 |
| 释放信号量 | void up(struct semaphore \* sem); |

#### Example

static ssize\_t globarl\_var(struct file \*file, const char \_\_user \*ubuf, size\_t count,loff\_t \*offp)

{

//试图获得信号量，用可被信号打断方式

if(down\_interruptible(&sema) < 0){

return -ERESTARTSYS;

}

//对共享资源(global\_var)进行操作

if(copy\_from\_user(&global\_var, buf, sizeof(int))) {

//失败也要进行释放信号量，要不死锁了

up(&sema);

return -EFAULT;

}

//成功释放信号量

up(&sema);

return sizeof(int);

}

### POSIX信号量

#### 无名信号量

1. 使用场景

相关进程（父子进程），同一进程的子线程间

1. 系统函数

|  |  |
| --- | --- |
| **头文件** | #include <semaphore.h> |
| **初始化** | int sem\_init(sem\_t \*sem, int pshared, unsigned int value);  1) pshared ==0 用于同一进程中多线程的同步； 2)若pshared>0 用于多个相关进程间的同步（即由fork产生的） |
| **取值** | int sem\_getvalue(sem\_t \*sem, int \*sval);   1) 返回0  2) 返回阻塞在该信号量上的进程或线程数目 |
| **申请（阻塞）** | int sem\_wait(sem\_t \*sem);     // 这是一个阻塞的函数  测试所指定信号量的值,它的操作是原子的, 若sem>0，那么它减1并立即返回。 若sem==0，则睡眠直到sem>0，此时立即减1，然后返回。 |
| **申请（非阻塞）** | int sem\_trywait(sem\_t \*sem);   // 非阻塞的函数 其他的行为和sem\_wait一样，除了： 若sem==0，不是睡眠，而是返回一个错误EAGAIN。 |
| **释放** | int sem\_post(sem\_t \*sem);  把指定的信号量sem的值加1;  呼醒正在等待该信号量的任意线程。  注意：在这些函数中，只有sem\_post是信号安全的函数，它是可重入函数 |
| **销毁** | int sem\_destroy(sem\_t \*sem); destroy an unnamed semaphore |

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <semaphore.h> | |
| **函数原型** | int sem\_init(sem\_t \*sem, int pshared, unsigned value); | |
| **参数** | sem | refer to a semaphore |
| pshared | If the pshared argument has a non-zero value, then the semaphore is shared between processes;  If the pshared argument is zero, then the semaphore is shared between threads of the process; |
| value | The value of the initialized semaphore shall be value |
| **返回值** | Upon successful completion, the sem\_init() function shall initialize the semaphore in sem. Otherwise, it shall return -1 and set errno to indicate the error. | |

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <semaphore.h> | |
| **函数原型** | int sem\_destroy(sem\_t \*sem); | |
| **参数** | sem | destroys the unnamed semaphore at the address pointed to by sem |
| **返回值** | em\_destroy() returns 0 on success; on error, -1 is returned, and errno is set to indicate the error. | |
| **说明** | Only a semaphore that has been initialised by sem\_init(3) should be destroyed using sem\_destroy().  Destroying a semaphore that other processes or threads are currently blocked on (in sem\_wait(3)) produces undefined behaviour.  Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialised using sem\_init(3). | |

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <semaphore.h> | |
| **函数原型** | int sem\_wait(sem\_t \*sem); | |
| **参数** | sem | decrements (locks) the semaphore pointed to by sem |
| **返回值** | return 0 on success; on error, the value of the semaphore is left unchanged, -1 is returned, and errno is set to indicate the error. | |
| **说明** | If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement | |

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <semaphore.h> | |
| **函数原型** | int sem\_trywait(sem\_t \*sem); | |
| **参数** | sem | decrements (locks) the semaphore pointed to by sem |
| **返回值** | return 0 on success; on error, the value of the semaphore is left unchanged, -1 is returned, and errno is set to indicate the error. | |
| **说明** | it is the same as sem\_wait(), except that if the decrement cannot be immediately performed, then call returns an error (errno set to EAGAIN) instead of blocking. | |

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <semaphore.h> | |
| **函数原型** | int sem\_timedwait(sem\_t \*sem, const struct timespec \*abs\_timeout); | |
| **参数** | sem | decrements (locks) the semaphore pointed to by sem |
| abs\_timeout | The abs\_timeout argument points to a structure that specifies an absolute timeout in seconds and nanoseconds since the Epoch (00:00:00, 1 January 1970). |
| **返回值** | return 0 on success; on error, the value of the semaphore is left unchanged, -1 is returned, and errno is set to indicate the error. | |
| **说明** | is the same as sem\_wait(), except that abs\_timeout specifies a limit on the amount of time that the call should block if the decrement cannot be immediately performed. | |

struct timespec {

time\_t tv\_sec; /\* Seconds \*/

long tv\_nsec; /\* Nanoseconds [0 .. 999999999] \*/

};

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <semaphore.h> | |
| **函数原型** | int sem\_post(sem\_t \*sem); | |
| **参数** | sem | increments (unlocks) the semaphore pointed to by sem. |
| **返回值** | sem\_post() returns 0 on success; on error, the value of the semaphore is left unchanged, -1 is returned, and errno is set to indicate the error. | |
| **说明** | The sem\_post() function shall be reentrant with respect to signals and may be invoked from a signal-catching function. | |

#### Example

#include <semaphore.h>

#include <stdio.h>

#include <errno.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <sys/mman.h>

int main(int argc, char \*\*argv)

{

int fd, i,count=0,nloop=10,zero=0,\*ptr;

sem\_t mutex;

//open a file and map it into memory

fd = open("log.txt",O\_RDWR|O\_CREAT,S\_IRWXU);

write(fd,&zero,sizeof(int));

ptr = mmap( NULL,sizeof(int),PROT\_READ | PROT\_WRITE,MAP\_SHARED,fd,0 );

close(fd);

/\* create, initialize semaphore \*/

if( sem\_init(&mutex,1,1) < 0) //

{

perror("semaphore initilization");

exit(0);

}

if (fork() == 0)

{ /\* child process\*/

for (i = 0; i < nloop; i++)

{

sem\_wait(&mutex);

printf("child: %d\n", (\*ptr)++);

sem\_post(&mutex);

}

exit(0);

}

/\* back to parent process \*/

for (i = 0; i < nloop; i++)

{

sem\_wait(&mutex);

printf("parent: %d\n", (\*ptr)++);

sem\_post(&mutex);

}

exit(0);

}

#### 有名信号量

**使用场景：**

既可以用于线程，也可以用于相关进程间，甚至是不相关进程。

有名信号量能在进程间共享的原因由于有名信号量的值是保存在文件中的，所以对于相关进程来说，子进程是继承了父进程的文件描述符，那么子进程所继承的文件描述符所指向的文件是和父进程一样的，当然文件里面保存的有名信号量值就共享了。

和无名信号量共享sem\_wait和sem\_post函数。

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <semaphore.h> | |
| **函数原型** | sem\_t \*sem\_open(const char \*name, int oflag);  sem\_t \*sem\_open(const char \*name, int oflag, mode\_t mode, unsigned int value); | |
| **参数** | name | 信号量文件名  这里的name不能写成/tmp/aaa.sem这样的格式，因为在linux下，sem都是创建在/dev/shm目录下。你可以将name写成“/mysem”或“mysem”，创建出来的文件都是“/dev/shm/sem.mysem”，千万不要写路径。也千万不要写“/tmp/mysem”之类的。 |
| oflag | 有O\_CREAT或O\_CREAT|EXCL两个取值；  当oflag = O\_CREAT时，若name指定的信号量不存在时，则会创建一个，而且后面的mode和value参数必须有效。若name指定的信号量已存在，则直接打开该信号量，同时忽略mode和value参数。  当oflag = O\_CREAT|O\_EXCL时，若name指定的信号量已存在，该函数会直接返回error。 |
| mode | 信号量的访问权限 |
| value | 信号量的初始化值 |
| **返回值** | On success, sem\_open() returns the address of the new semaphore; this address is used when calling other semaphore-related functions.  On error, sem\_open() returns SEM\_FAILED, with errno set to indicate the error. | |

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <semaphore.h> | |
| **函数原型** | int sem\_close(sem\_t \*sem); | |
| **参数** | sem | sem\_close() closes the named semaphore referred to by sem |
| **返回值** | On success sem\_close() returns 0; on error, -1 is returned, with errno set to indicate the error. | |

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <semaphore.h> | |
| **函数原型** | int sem\_unlink(const char \*name); | |
| **参数** | name | 信号量文件名 |
| **返回值** | Upon successful completion, the sem\_unlink() function shall return a value of 0. Otherwise, the semaphore shall not be changed and the function shall return a value of -1 and set errno to indicate the error. | |
| **说明** | 该函数会等待所有引用name指向的信号量都关闭（sem\_close）后，才会起作用，才会删除该信号量。  因为每个信号量有一个引用计数器记录当前的打开次数，sem\_unlink必须等待这个数为0时才能把name所指的信号灯从文件系统中删除。 | |

#### Example

**File1: server.c**

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

#include <semaphore.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#define SHMSZ 27

char SEM\_NAME[]= "vik";

int main()

{

char ch;

int shmid;

key\_t key;

char \*shm,\*s;

sem\_t \*mutex;

//name the shared memory segment

key = 1000;

//create & initialize semaphore

mutex = sem\_open(SEM\_NAME,O\_CREAT,0644,1);

if(mutex == SEM\_FAILED)

{

perror("unable to create semaphore");

sem\_unlink(SEM\_NAME);

exit(-1);

}

//create the shared memory segment with this key

shmid = shmget(key,SHMSZ,IPC\_CREAT|0666);

if(shmid<0)

{

perror("failure in shmget");

exit(-1);

}

//attach this segment to virtual memory

shm = shmat(shmid,NULL,0);

//start writing into memory

s = shm;

for(ch='A';ch<='Z';ch++)

{

sem\_wait(mutex);

\*s++ = ch;

sem\_post(mutex);

}

//the below loop could be replaced by binary semaphore

while(\*shm != '\*')

{

sleep(1);

}

sem\_close(mutex);

sem\_unlink(SEM\_NAME);

shmctl(shmid, IPC\_RMID, 0);

exit(0);

}

**File 2: client.c**

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <stdio.h>

#include <semaphore.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#define SHMSZ 27

char SEM\_NAME[]= "vik";

int main()

{

char ch;

int shmid;

key\_t key;

char \*shm,\*s;

sem\_t \*mutex;

//name the shared memory segment

key = 1000;

//create & initialize existing semaphore

mutex = sem\_open(SEM\_NAME,0,0644,0);

if(mutex == SEM\_FAILED)

{

perror("reader:unable to execute semaphore");

sem\_close(mutex);

exit(-1);

}

//create the shared memory segment with this key

shmid = shmget(key,SHMSZ,0666);

if(shmid<0)

{

perror("reader:failure in shmget");

exit(-1);

}

//attach this segment to virtual memory

shm = shmat(shmid,NULL,0);

//start reading

s = shm;

for(s=shm;\*s!=NULL;s++)

{

sem\_wait(mutex);

putchar(\*s);

sem\_post(mutex);

}

//once done signal exiting of reader:This can be replaced by another semaphore

\*shm = '\*';

sem\_close(mutex);

shmctl(shmid, IPC\_RMID, 0);

exit(0);

}

### SYSTEM V信号量

#### 信号量结构

内核为每个信号量集维护一个信号量结构体，可在<sys/sem.h>找到该定义：

struct semid\_ds {

struct ipc\_perm sem\_perm; /\* 信号量集的操作许可权限 \*/

struct sem \*sem\_base; /\* 某个信号量sem结构数组的指针，当前信号量集

中的每个信号量对应其中一个数组元素 \*/

ushort sem\_nsems; /\* sem\_base 数组的个数 \*/

time\_t sem\_otime; /\* 最后一次成功修改信号量数组的时间 \*/

time\_t sem\_ctime; /\* 成功创建时间 \*/

};

struct sem {

ushort semval; /\* 信号量的当前值 \*/

short sempid; /\* 最后一次返回该信号量的进程ID 号 \*/

ushort semncnt; /\* 等待semval大于当前值的进程个数 \*/

ushort semzcnt; /\* 等待semval变成0的进程个数 \*/

};

#### 系统函数

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h> | |
| **函数原型** | int semget(key\_t key, int nsems, int oflag) | |
| **参数** | Key | Key |
| Nsems | 信号量数量 |
| Oflag | 访问权限 |
| **返回值** | If successful, the return value will be the semaphore set identifier (a nonnegative integer), otherwise -1 is returned, with errno indicating the error. | |

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h> | |
| **函数原型** | int semop(int semid, struct sembuf \*opsptr, size\_t nops); | |
| **参数** | Semid | Semaphore set ID |
| opsptr | 指向sembuf的数组指针 |
| nops | Sembuf的个数 |
| **返回值** | Upon successful completion, semop() shall return 0; otherwise, it shall return -1 and set errno to indicate the error. | |

|  |  |  |
| --- | --- | --- |
| **头文件** | #include <sys/types.h>  #include <sys/ipc.h>  #include <sys/sem.h> | |
| **函数原型** | int semctl(int semid, int semnum, int cmd, ../\* union semun arg \*/); | |
| **参数** | semid | Sem ID |
| semnum | 信号量在集合中的序号 |
| Cmd | 决定了是否有第四个参数  IPC\_STAT  IPC\_SET  IPC\_RMID  IPC\_INFO(linux-specific)  SEM\_INFO(linux-specific)  SEM\_STAT(linux-specific)  GETALL  GETPID  GETVAL  GETZCNT  SETALL  SETVAL |
| **返回值** | On failure | -1, set errno |
| GETCNT | Value of semncnt等待信号量值大于当前值的进程数 |
| GETPID | 最后一个操作semnum-th信号量的PID |
| GETVAL | Value of semval信号量值 |
| GETZCNT | Value of semzcnt等待信号量值变为0的进程数 |
| IPC\_INFO | 忽略semnum，copy内核空间的semid\_ds到arg.buf |
| SEM\_INFO | 同IPC\_INFO |
| SEM\_STAT | Id of Semaphore set whose index was given in semid. |
| 其他cmd | 0 |

 第四个参数：

union semun {

int val; // cmd == SETVAL

struct semid\_ds \*buf; // cmd == IPC\_SET或者 cmd == IPC\_STAT

ushort \*array; // cmd == SETALL，或 cmd = GETALL

struct seminfo \*\_\_buf; // cmd == IPC\_INFO(linux-specific)

}

#### Example：

#include <sys/types.h>

#include <stdio.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <sys/stat.h>

#include <errno.h>

#include <unistd.h>

#include <stdlib.h>

#include <pwd.h>

#include <fcntl.h>

#include <limits.h>

...

key\_t semkey;

int semid, pfd, fv;

struct sembuf sbuf;

char \*lgn;

char filename[PATH\_MAX+1];

struct stat outstat;

struct passwd \*pw;

...

/\* Get unique key for semaphore. \*/

if ((semkey = ftok("/tmp", 'a')) == (key\_t) -1) {

perror("IPC error: ftok"); exit(1);

}

/\* Get semaphore ID associated with this key. \*/

if ((semid = semget(semkey, 0, 0)) == -1) {

/\* Semaphore does not exist - Create. \*/

if ((semid = semget(semkey, 1, IPC\_CREAT | IPC\_EXCL | S\_IRUSR |

S\_IWUSR | S\_IRGRP | S\_IWGRP | S\_IROTH | S\_IWOTH)) != -1)

{

/\* Initialize the semaphore. \*/

sbuf.sem\_num = 0;

sbuf.sem\_op = 2; /\* This is the number of runs without queuing. \*/

sbuf.sem\_flg = 0;

if (semop(semid, &sbuf, 1) == -1) {

perror("IPC error: semop"); exit(1);

}

}

else if (errno == EEXIST) {

if ((semid = semget(semkey, 0, 0)) == -1) {

perror("IPC error 1: semget"); exit(1);

}

}

else {

perror("IPC error 2: semget"); exit(1);

}

}

...

sbuf.sem\_num = 0;

sbuf.sem\_op = -1;

sbuf.sem\_flg = SEM\_UNDO;

if (semop(semid, &sbuf, 1) == -1) {

perror("IPC Error: semop"); exit(1);

}

#### 参考

<http://blog.csdn.net/qinxiongxu/article/details/7830537>

<http://blog.csdn.net/weed_hz/article/details/8965733>

### 读-写信号量

**跟自旋锁一样，信号量也有区分读-写信号量之分**

如果一个读写信号量当前没有被写者拥有并且也没有写者等待读者释放信号量，那么任何读者都可以成功获得该读写信号量；否则，读者必须被挂起直到写者释放该信号量。

如果一个读写信号量当前没有被读者或写者拥有并且也没有写者等待该信号量，那么一个写者可以成功获得该读写信号量，否则写者将被挂起，直到没有任何访问者。因此，写者是排他性的，独占性的。

读写信号量有两种实现:

一种是通用的，不依赖于硬件架构，因此，增加新的架构不需要重新实现它，但缺点是性能低，获得和释放读写信号量的开销大；

另一种是架构相关的，因此性能高，获取和释放读写信号量的开销小，但增加新的架构需要重新实现。在内核配置时，可以通过选项去控制使用哪一种实现。

读写信号量的相关API有：

|  |  |
| --- | --- |
| **API** | **说明** |
| DECLARE\_RWSEM(name) | 该宏声明一个读写信号量name并对其进行初始化。 |
| void init\_rwsem(struct rw\_semaphore \*sem); | 该函数对读写信号量sem进行初始化。 |
| void down\_read(struct rw\_semaphore \*sem); | 读者调用该函数来得到读写信号量sem。该函数会导致调用者睡眠，因此只能在进程上下文使用。 |
| int down\_read\_trylock(struct rw\_semaphore \*sem); | 该函数类似于down\_read，只是它不会导致调用者睡眠。它尽力得到读写信号量sem，如果能够立即得到，它就得到该读写信号量，并且返回1，否则表示不能立刻得到该信号量，返回0。因此，它也可以在中断上下文使用。 |
| void down\_write(struct rw\_semaphore \*sem); | 写者使用该函数来得到读写信号量sem，它也会导致调用者睡眠，因此只能在进程上下文使用。 |
| int down\_write\_trylock(struct rw\_semaphore \*sem); | 该函数类似于down\_write，只是它不会导致调用者睡眠。该函数尽力得到读写信号量，如果能够立刻获得，就获得该读写信号量并且返回1，否则表示无法立刻获得，返回0。它可以在中断上下文使用。 |
| void up\_read(struct rw\_semaphore \*sem); | 读者使用该函数释放读写信号量sem。它与down\_read或down\_read\_trylock配对使用。如果down\_read\_trylock返回0，不需要调用up\_read来释放读写信号量，因为根本就没有获得信号量。 |
| void up\_write(struct rw\_semaphore \*sem); | 写者调用该函数释放信号量sem。它与down\_write或down\_write\_trylock配对使用。如果down\_write\_trylock返回0，不需要调用up\_write，因为返回0表示没有获得该读写信号量。 |
| void downgrade\_write(struct rw\_semaphore \*sem); | 该函数用于把写者降级为读者，这有时是必要的。因为写者是排他性的，因此在写者保持读写信号量期间，任何读者或写者都将无法访问该读写信号量保护的共享资源，对于那些当前条件下不需要写访问的写者，降级为读者将使得等待访问的读者能够立刻访问，从而增加了并发性，提高了效率。 |

读写信号量适于在读多写少的情况下使用，在linux内核中对进程的内存映像描述结构的访问就使用了读写信号量进行保护。

究竟什么时候使用自旋锁什么时候使用信号量，下面给出建议的方案

当对低开销、短期、中断上下文加锁，优先考虑自旋锁；

当对长期、持有锁需要休眠的任务，优先考虑信号量。

## Socket

# Linux中的锁

## 自旋锁

UP: 单CPU

SMP: 多CPU（symetric multi-processer）

### 概念

如果内核控制路径发现自旋锁“开着”（可以获取），就获取锁并继续自己的执行。相反，如果内核控制路径发现锁由运行在另一个CPU上的内核控制路径“锁着”，就在原地“旋转”，反复执行一条紧凑的循环检测指令，直到锁被释放。 自旋锁是循环检测“忙等”，即等待时内核无事可做（除了浪费时间），进程在CPU上保持运行，所以它保护的临界区必须小，且操作过程必须短。不过，自旋锁通常非常方便，因为很多内核资源只锁1毫秒的时间片段，所以等待自旋锁的释放不会消耗太多CPU的时间。

### 原子操作

为保证操作的原子性，自旋锁需要阻止代码运行过程中出现的干扰，这些干扰包括：

* **中断**

分为硬件中断和软件中断，中断处理程序有可能会访问临界区，这样就破坏了操作的原子性，所以需要spinlock来保护临界区。

* **内核抢占**

Linux 2.6以后支持内核抢占，且可配置。这使UP系统和SMP类似，会出现内核态下的并发。这种情况下进入临界区就需要避免因抢占造成的并发，所以解决的方法就是在加锁时禁用抢占（preempt\_disable(); ），在开锁时开启抢占（preempt\_enable();注意此时会执行一次抢占调度。

* **其他处理器访问同一临界区**

在SMP系统中，多个物理处理器同时工作，导致可能有多个进程物理上的并发。这样就需要在内存加一个标志，每个需要进入临界区的代码都必须检查这个标志，看是否有进程已经在这个临界区中。这种情况下检查标志的代码也必须保证原子和快速，这就要求必须精细地实现，正常情况下每个构架都有自己的汇编实现方案，保证检查的原子性。

这需要解决以下两个问题：

（1）你如何保证在SMP下其他处理器不会同时访问同一个的标志呢？（也就是标志的独占访问）

（2）必须保证每个处理器都不会去读取高速缓存而是真正的内存中的标志（可以实现，编程上可以用volatile）

要根本解决这个问题，需要在芯片底层实现物理上的内存地址独占访问，并且在实现上使用特殊的汇编指令访问。请看参考资料中对于自旋锁的实现分析。以arm为例，从存在SMP的ARM构架指令集开始（V6、V7），采用LDREX和STREX指令实现真正的自旋等待。

### 自旋锁操作的组成

中断控制（仅在中断代码可能访问临界区时需要）

抢占控制（仅存在于可抢占内核中需要）

自旋锁标志控制（仅SMP系统需要）

### 自旋锁变体的使用规则

不论是抢占式UP、非抢占式UP还是SMP系统，只要在某类中断代码可能访问临界区，就需要控制中断，保证操作的原子性。所以这个和模块代码中临界区的访问还有关系，是否可能在中断中操作临界区，只有程序员才知道。所以自旋锁API中有针对不同中断类型的自旋锁变体：

| **应用场景** | **使用API** |
| --- | --- |
| 不会在任何中断例程中操作临界区 | static inline void spin\_lock(spinlock\_t \*lock)  static inline void spin\_unlock(spinlock\_t \*lock) |
| 如果在软件中断中操作临界区 | static inline void spin\_lock\_bh(spinlock\_t \*lock)  static inline void spin\_unlock\_bh(spinlock\_t \*lock)  bh代表bottom half，也就是中断中的底半部，因内核中断的底半部一般通过软件中断（tasklet等）来处理而得名。 |
| 如果在硬件中断中操作临界区 | static inline void spin\_lock\_irq(spinlock\_t \*lock)  static inline void spin\_unlock\_irq(spinlock\_t \*lock) |
| 如果在控制硬件中断的时候需要同时保存之前的中断状态 | spin\_lock\_irqsave(lock, flags)  static inline void spin\_unlock\_irqrestore(spinlock\_t \*lock, unsigned long flags) |

另一个版本

|  |  |
| --- | --- |
| **上下文** | **API** |
| 如果被保护的共享资源只在进程上下文访问和软中断（包括tasklet、timer）上下文访问，那么当在进程上下文访问共享资源时，可能被软中断打断，从而可能进入软中断上下文来对被保护的共享资源访问， | 因此对于这种情况，对共享资源的访问必须使用spin\_lock\_bh和spin\_unlock\_bh来保护。当然使用spin\_lock\_irq和spin\_unlock\_irq以及spin\_lock\_irqsave和spin\_unlock\_irqrestore也可以，它们失效了本地硬中断，失效硬中断隐式地也失效了软中断。但是使用spin\_lock\_bh和spin\_unlock\_bh是最恰当的，它比其他两个快。 |
| 如果被保护的共享资源只在两个或多个tasklet或timer上下文访问， | 那么对共享资源的访问仅需要用spin\_lock和spin\_unlock来保护，不必使用\_bh版本，因为当tasklet或timer运行时，不可能有其他tasklet或timer在当前CPU上运行。 |
| 如果被保护的共享资源只在一个tasklet或timer上下文访问，那么不需要任何自旋锁保护，因为同一个tasklet或timer只能在一个CPU上运行，即使是在SMP环境下也是如此。 | 不需要任何自旋锁保护  实际上tasklet在调用tasklet\_schedule标记其需要被调度时已经把该tasklet绑定到当前CPU，因此同一个tasklet决不可能同时在其他CPU上运行。timer也是在其被使用add\_timer添加到timer队列中时已经被帮定到当前CPU，所以同一个timer绝不可能运行在其他CPU上。当然同一个tasklet有两个实例同时运行在同一个CPU就更不可能了。 |
| 如果被保护的共享资源只在一个软中断（tasklet和timer除外）上下文访问， | 那么这个共享资源需要用spin\_lock和spin\_unlock来保护，因为同样的软中断可以同时在不同的CPU上运行。 |
| 如果被保护的共享资源在两个或多个软中断上下文访问， | 那么这个共享资源当然更需要用spin\_lock和spin\_unlock来保护，不同的软中断能够同时在不同的CPU上运行。 |
| 如果被保护的共享资源在软中断（包括tasklet和timer）或进程上下文和硬中断上下文访问， | 那么在软中断或进程上下文访问期间，可能被硬中断打断，从而进入硬中断上下文对共享资源进行访问，因此，在进程或软中断上下文需要使用spin\_lock\_irq和spin\_unlock\_irq来保护对共享资源的访问。 |
| 而在中断处理句柄中使用什么版本，需依情况而定，如果只有一个中断处理句柄访问该共享资源， | 那么在中断处理句柄中仅需要spin\_lock和spin\_unlock来保护对共享资源的访问就可以了。因为在执行中断处理句柄期间，不可能被同一CPU上的软中断或进程打断。 |
| 但是如果有不同的中断处理句柄访问该共享资源， | 那么需要在中断处理句柄中使用spin\_lock\_irq和spin\_unlock\_irq来保护对共享资源的访问。 |
| 在使用spin\_lock\_irq和spin\_unlock\_irq的情况下，完全可以用spin\_lock\_irqsave和spin\_unlock\_irqrestore取代，那具体应该使用哪一个也需要依情况而定， | 如果可以确信在对共享资源访问前中断是使能的，那么使用spin\_lock\_irq更好一些。因为它比spin\_lock\_irqsave要快一些，但是如果你不能确定是否中断使能，那么使用spin\_lock\_irqsave和spin\_unlock\_irqrestore更好，因为它将恢复访问共享资源前的中断标志而不是直接使能中断。 |
| 当然，有些情况下需要在访问共享资源时必须中断失效，而访问完后必须中断使能， | 这样的情形使用spin\_lock\_irq和spin\_unlock\_irq最好。 |
| spin\_lock用于阻止在不同CPU上的执行单元对共享资源的同时访问以及不同进程上下文互相抢占导致的对共享资源的非同步访问，而中断失效和软中断失效却是为了阻止在同一CPU上软中断或中断对共享资源的非同步访问。 |  |

See <http://blog.chinaunix.net/uid-26990992-id-3264808.html>

## Tasklet和软中断



## 互斥锁

# 进程上下文和中断上下文

处理器总处于以下三种状态之一：  
１、内核态，运行于进程上下文，内核代表进程运行于内核空间；  
２、内核态，运行于中断上下文，内核代表硬件运行于内核空间；  
３、用户态，运行于用户空间。

## 进程上下文

LINUX完全注释中的一段话：

当一个进程在执行时,CPU的所有寄存器中的值、进程的状态以及堆栈中的内容被称 为该进程的上下文。当内核需要切换到另一个进程时，它需要保存当前进程的 所有状态，即保存当前进程的上下文，以便在再次执行该进程时，能够必得到切换时的状态执行下去。在LINUX中，当前进程上下文均保存在进程的任务数据结构中。在发生中断时,内核就在被中断进程的上下文中，在内核态下执行中断服务例程。但同时会保留所有需要用到的资源，以便中断服务结束时能恢复被中断进程的执行

上下文context： 上下文简单说来就是一个环境，相对于进程而言，就是进程执行时的环境。具体来说就是各个变量和数据，包括所有的寄存器变量、进程打开的文件、内存信息等。

一个进程的上下文可以分为三个部分:

1) 用户级上下文: 正文、数据、用户堆栈以及共享存储区；

2) 寄存器上下文: 通用寄存器、程序寄存器(IP)、处理器状态寄存器(EFLAGS)、栈指针(ESP)；

3) 系统级上下文: 进程控制块task\_struct、内存管理信息(mm\_struct、vm\_area\_struct、pgd、pte)、内核栈。

当发生进程调度时，进行进程切换就是上下文切换(context switch).操作系统必须对上面提到的全部信息进行切换，新调度的进程才能运行。

而系统调用进行的模式切换(mode switch)与进程切换比较起来，容易很多，而且节省时间，因为模式切换最主要的任务只是切换进程寄存器上下文的切换。

## 中断上下文

When executing an interrupt handler or bottom half, the kernel is in interrupt context.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Sleep | Current macro(宏) | Stack size |
| Process context  进程上下文 | 可以 | 指向当前process | 2pages on 32-bit architecture(8KB)  4pages on 64-bit architecture(16KB) |
| Interrupt context  中断上下文 | 不可以 | 没有进程背景backing process  所以和该宏无关 | 过去：和process共享  现在：独自拥有1page size(4KB) |

一个进程进入内核空间的途径

System calls and exception handlers are well-defined interfaces into the kernel. A process can begin executing in kernel-space only through one of these interfaces -- all access to the kernel is through these interfaces.

# 可重入函数

## 特点

可以在多个CPU上并行地执行，而不互相影响，每个函数都能正确地执行。

一个可重入的函数简单来说就是可以被中断的函数，也就是说，可以在这个函数执行的任何时刻中断它，转入OS调度下去执行另外一段代码，而返回控制时不会出现什么错误

* 可并行多个
* 可中断

## 应用场景

多任务环境（多线程，多进程调用）

## 如何实现

* 不使用共享的数据
* 使用共享数据时，通过同步方式（关中断，信号量等）访问共享数据避免被意外篡改

# 编程

## C语言

### 可变参函数

#include <stdarg.h>

void va\_start(va\_list ap, last); //初始化ap（变参的首地址），last是最后一个固定参数

**注意:由于参数的地址用于va\_start宏,所以参数不能声明为寄存器变量或作为函数或数组类型.**(Because the address of this parameter may be used in the va\_start() macro, it should not be declared as a register variable, or as a function or an array type.)

type va\_arg(va\_list ap, type); //ap是某个参数的地址，将其按照type进行转换，从而获得ap的值。

void va\_end(va\_list ap); //将ap设为NULL，即ap=(void\*)0;

例子：

#include <stdio.h>

#include <stdarg.h>

void foo(char \*fmt, ...) {

va\_list ap;

int d;

char c, \*s;

va\_start(ap, fmt);

while (\*fmt)

switch(\*fmt++) {

case 's': /\* string \*/

s = va\_arg(ap, char \*);

printf("string %s\n", s);

break;

case 'd': /\* int \*/

d = va\_arg(ap, int);

printf("int %d\n", d);

break;

case 'c': /\* char \*/

/\* need a cast here since va\_arg only takes fully promoted types \*/

c = (char) va\_arg(ap, int);

printf("char %c\n", c);

break;

}

va\_end(ap);

}

**可变参宏**

#define fnAdd(...) \_fnAdd(0,**\_\_VA\_ARGS\_\_**)

然后再定义一个可变参函数）\_fnAdd即可。

### Exec族函数

SYNOPSIS

#include <unistd.h>

extern char \*\*environ;

int execl(const char \*path, const char \*arg, ...);

int execlp(const char \*file, const char \*arg, ...);

int execle(const char \*path, const char \*arg,

..., char \* const envp[]);

int execv(const char \*path, char \*const argv[]);

int execvp(const char \*file, char \*const argv[]);

RETURN VALUE

If any of the exec() functions returns, an error will have occurred. The return value is -1, and the global variable errno will be set to indicate the error.

exec函数族的作用是**根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件**。这里的可执行文件既可以是**二进制文件**，也可以是任何Linux下可执行的**脚本文件**。

字母l是指函数的参数是以list的形式传入

字母v是指函数的参数是以vector形式传入

字母p是指在环境变量PATH的目录里去查找要执行的可执行文件。

字母e是指可以在函数中定义一组环境变量，替代默认的环境变量

### fork和vfork

通过fork创建的子进程，其子进程的内存空间是从父进程拷贝而来，有时这种拷贝会是一种浪费，比如在子进程中调用了exec后，父进程的内存空间就完全被抹掉了，所以就有了“写时拷贝”，即只有子进程要修改进程空间时才会拷贝。

而vfork则完全不拷贝父进程的内存空间，子进程共享父进程的内存空间比如堆栈等内存，这样一来子进程就不能随意改变栈上变量，不能使用exec和\_exit之外的其他函数。所以，为了增加了一个限制，一旦vfork后，父进程就会被挂起，直到子进程有了自己的进程空间（exec\*\*或\_exit）。

### 静态变量

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 存储空间 | 生命周期 | 作用域 | 初始值 |
| 静态局部变量 | 静态（或全局）存储区 | 整个程序的运行期间 | 本函数内 | 用户不初始化的话，系统自动赋初值0；  始化表达式必须是一个常量或者常量表达式 |
| 静态全局变量 | 同上 | 同上 | 定义该变量的源文件 |  |
| 局部变量（auto） | 函数栈上 | 函数的运行期间 | 函数内 | 值随机 |
| 非静态全局变量 | 静态存储区 | 整个程序运行期间 | 全部的源文件 |  |

**注意**

* 所有的全局变量都是静态变量，而局部变量只有定义时加上类型修饰符static，才为局部静态变量
* 静态变量并不是说其就不能改变值，不能改变值的量叫常量。
* 局部静态变量占用内存时间较长，并且可读性差，因此，除非必要，尽量避免使用局部静态变量。
* 把局部变量改变为静态变量后是改变了它的存储方式，即改变了它的生存期。
* 把全局变量改变为静态变量后是改变了它的作用域，限制了它的使用范围。
* 如果将函数声明为Static类型，则该函数名除了对该函数声明所在的文件可见外，其他文件均无法访问。

**使用原则：**

如果我们需要一个可重入的函数，那么，我们一定要避免函数中使用static变量，这种函数中的static变量，使用原则是，能不用尽量不用。

有些时候，在函数中是必须要使用static变量的，比如当某函数的返回值为指针类型时，则必须是static的局部变量的地址作为返回值，若为auto类型，则返回为错指针。

See <http://www.cnblogs.com/dongzhiquan/archive/2009/07/21/1994792.html>

### 位域

使用位域的主要目的是压缩存储，其大致规则为：

1) 如果相邻位域字段的类型相同，且其位宽之和小于类型的sizeof大小，则后面的字段将紧邻前一个字段存储，直到不能容纳为止；

2) 如果相邻位域字段的类型相同，但其位宽之和大于类型的sizeof大小，则后面的字段将从新的存储单元开始，其偏移量为其类型大小的整数倍；

3) 如果相邻的位域字段的类型不同，则各编译器的具体实现有差异，VC6采取不压缩方式，Dev-C++采取压缩方式；

4) 如果位域字段之间穿插着非位域字段，则不进行压缩；

5) 整个结构体的总大小为最宽基本类型成员大小的整数倍。

struct bs

{

    unsigned a:4;

    unsigned :0;

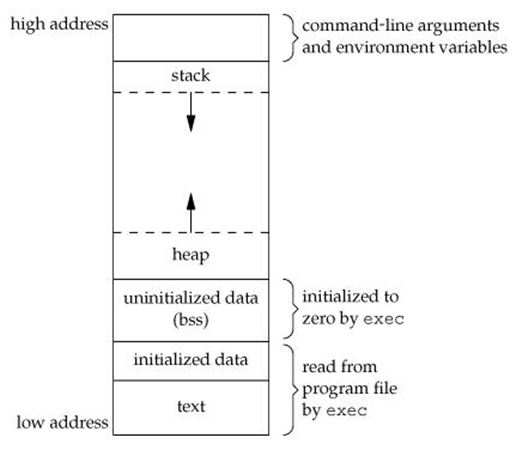
    unsigned b:4;

    unsigned c:4;

}

## Linux 程序存储空间

|  |  |  |
| --- | --- | --- |
| 分区 | 说明 | 内存分配方式 |
| BSS(block started by symbol) | 用来存放程序中未初始化的全局变量的一块内存区域 | 静态内存分配 |
| 数据段（code segment/text segment） | 指用来存放程序已经初始化的全局变量的一块内存区域 | 静态内存分配 |
| 代码段 | 指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读，某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，比如字符串常量等 |  |
| 栈（stack） | 是用户存放程序临时创建的局部变量，也就是我们函数括弧{}中定义的比变量（但不包括static声明的静态变量，static意味着在数据段中存放的变量）。除此之外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且等到调用结束后，函数的返回值也会被存放回栈中。由于栈先进后出的特点，所以栈特别方便用来保存/恢复调用现场。从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区。 |  |
| 堆（heap） | 用于存放进程运行中动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用malloc等函数分配内存的时候，新分配的内存就被动态添加到堆上（堆被扩张）；当利用free等函数释放内存的时候，被释放的内存从堆中被剔除（堆被缩减）。 |  |



## C++语言

## JAVA

# 网络

## 编程

**Socket编程**

**SYNOPSIS**

**#include <sys/types.h>**

**#include <sys/socket.h>**

**int socket(int domain, int type, int protocol);**

Server & Client

Server

Client

socket

bind

listen

socket

connect

Server

Accept

## 工具

|  |  |
| --- | --- |
| 路由配置 |  |
| 防火墙 |  |
|  |  |
| 端口查看 | netstat |
| 域名查看 | nslookup |

## 协议

### ARP/RARP

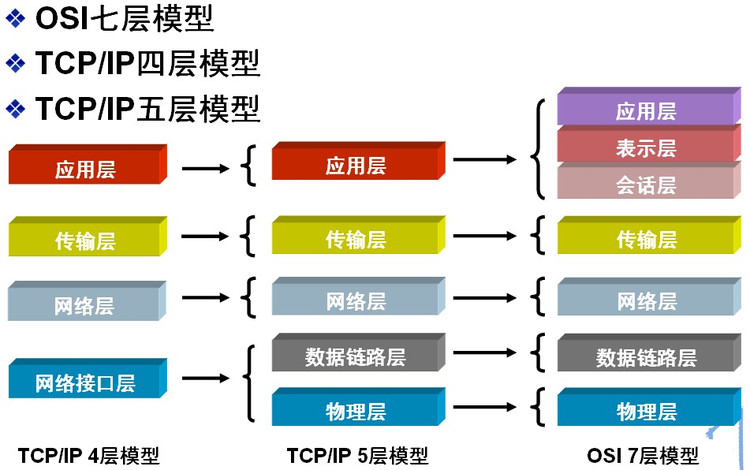
ARP是知道对方[IP](https://www.baidu.com/s?wd=IP&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3myf4nHubPHT3ujI-nj010ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHmzPW6LP1Dz)来请求对方[MAC](https://www.baidu.com/s?wd=MAC&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3myf4nHubPHT3ujI-nj010ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHmzPW6LP1Dz)地址，比如上网的时候网关是192.168.1.1，但是不知道网关的[MAC](https://www.baidu.com/s?wd=MAC&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3myf4nHubPHT3ujI-nj010ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHmzPW6LP1Dz)，这时就发送ARP广播包，在全网“呼叫”192.168.1.1，当网关听到有机器“嚎叫”就返回一个数据包告诉你“我的[MAC](https://www.baidu.com/s?wd=MAC&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3myf4nHubPHT3ujI-nj010ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHmzPW6LP1Dz)是XXXXXXXXX”然后双方就可以通信了，在你“嚎叫”的时候，按照规则，你需要把你的[IP](https://www.baidu.com/s?wd=IP&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3myf4nHubPHT3ujI-nj010ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHmzPW6LP1Dz)和MAC地址广播出去，否则网关只听“嚎叫”不见地址，一样联系不到你~~~~~网内其他主机则把你的“嚎叫”当作耳边风，不响应！  
  
[RARP](https://www.baidu.com/s?wd=RARP&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3myf4nHubPHT3ujI-nj010ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHmzPW6LP1Dz)是知道对方MAC来查询[IP](https://www.baidu.com/s?wd=IP&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3myf4nHubPHT3ujI-nj010ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHmzPW6LP1Dz)地址，基本上还是以上的过程，只不过你“嚎叫”的是MAC地址而已，在一般应用中很少有知道MAC来查询IP的现象，在网络软件设置的多是IP，不是MAC！！！  
对了这里补充下，[RARP](https://www.baidu.com/s?wd=RARP&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3myf4nHubPHT3ujI-nj010ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHmzPW6LP1Dz)需要有服务器支持才能响应，不是任何一个主机都能响应[RARP](https://www.baidu.com/s?wd=RARP&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3myf4nHubPHT3ujI-nj010ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EnHmzPW6LP1Dz)的，当RARP服务器收到请求，会搜索自己本地列表，发现对应的条目，然后给才能你分配地址，也就是说，RARP是个手动维护的协议，RARP在无盘上应用较多，平时极少碰到~~~

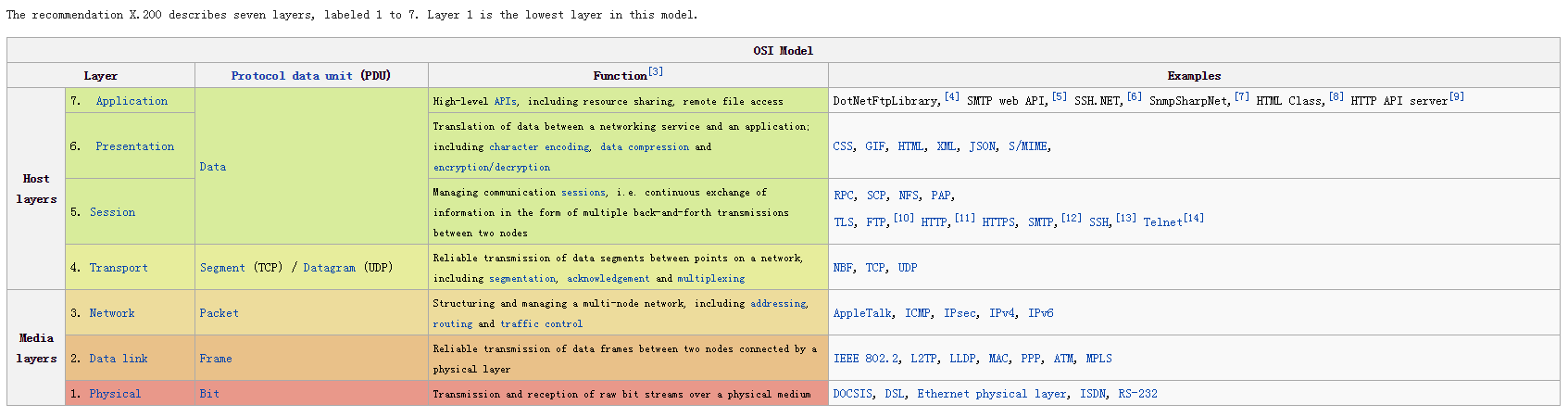
需要注意的是这个与DHCP协议是不同的~~~~

## TCPIP

### OSI model

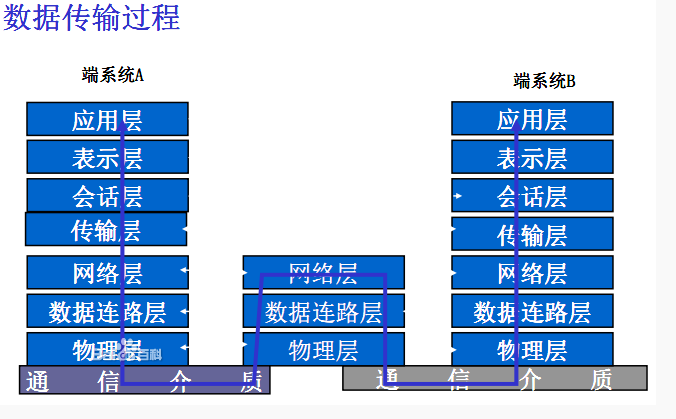
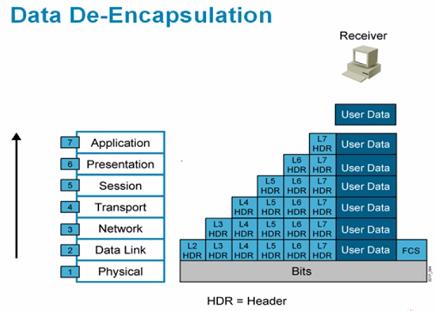
**Open Systems Interconnection model**

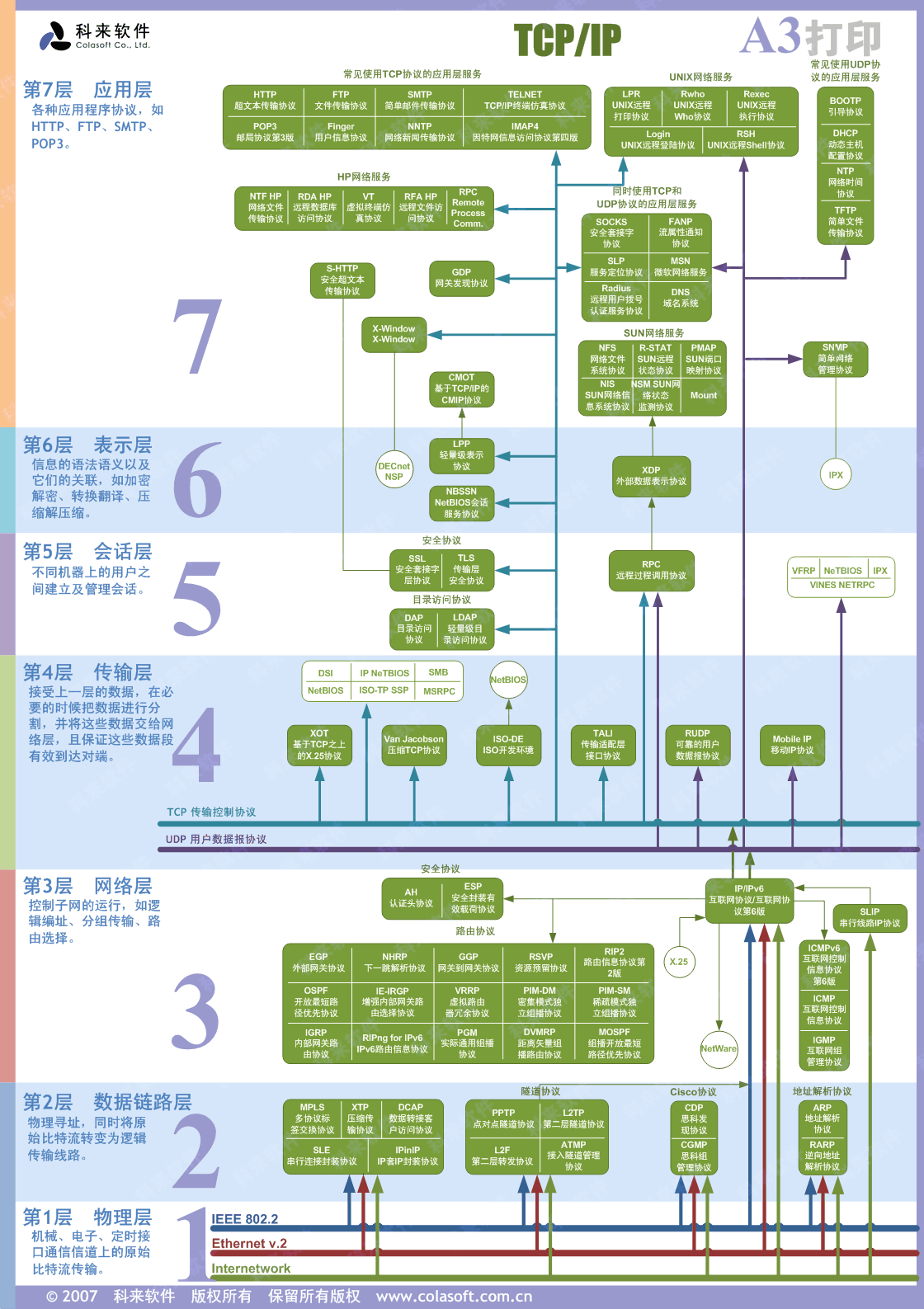




|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OSI Model** | | | | |
| **Layer** | | [**Protocol data unit**](https://en.wikipedia.org/wiki/Protocol_data_unit)**(PDU)** | **Function**[[3]](https://en.wikipedia.org/wiki/OSI_model#cite_note-3) | **Examples** |
| **Host layers** | 7. [Application](https://en.wikipedia.org/wiki/Application_layer) | [Data](https://en.wikipedia.org/wiki/Data_(computing)) | High-level [APIs](https://en.wikipedia.org/wiki/API), including resource sharing, remote file access | DotNetFtpLibrary,[[4]](https://en.wikipedia.org/wiki/OSI_model#cite_note-4) SMTP web API,[[5]](https://en.wikipedia.org/wiki/OSI_model#cite_note-5) SSH.NET,[[6]](https://en.wikipedia.org/wiki/OSI_model#cite_note-6) SnmpSharpNet,[[7]](https://en.wikipedia.org/wiki/OSI_model#cite_note-7) HTML Class,[[8]](https://en.wikipedia.org/wiki/OSI_model#cite_note-8) HTTP API server[[9]](https://en.wikipedia.org/wiki/OSI_model#cite_note-9) |
| 6. [Presentation](https://en.wikipedia.org/wiki/Presentation_layer) | Translation of data between a networking service and an application; including [character encoding](https://en.wikipedia.org/wiki/Character_encoding), [data compression](https://en.wikipedia.org/wiki/Data_compression) and[encryption/decryption](https://en.wikipedia.org/wiki/Encryption) | [CSS](https://en.wikipedia.org/wiki/CSS), [GIF](https://en.wikipedia.org/wiki/GIF), [HTML](https://en.wikipedia.org/wiki/HTML), [XML](https://en.wikipedia.org/wiki/XML), [JSON](https://en.wikipedia.org/wiki/JSON), [S/MIME](https://en.wikipedia.org/wiki/S/MIME), |
| 5. [Session](https://en.wikipedia.org/wiki/Session_layer) | Managing communication [sessions](https://en.wikipedia.org/wiki/Session_(computer_science)), i.e. continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes | [RPC](https://en.wikipedia.org/wiki/Remote_Procedure_Call), [SCP](https://en.wikipedia.org/wiki/DECnet), [NFS](https://en.wikipedia.org/wiki/Network_File_System), [PAP](https://en.wikipedia.org/wiki/Password_authentication_protocol),  [TLS](https://en.wikipedia.org/wiki/Transport_Layer_Security), [FTP](https://en.wikipedia.org/wiki/FTP),[[10]](https://en.wikipedia.org/wiki/OSI_model#cite_note-10) [HTTP](https://en.wikipedia.org/wiki/HTTP),[[11]](https://en.wikipedia.org/wiki/OSI_model#cite_note-11) [HTTPS](https://en.wikipedia.org/wiki/HTTPS), [SMTP](https://en.wikipedia.org/wiki/SMTP),[[12]](https://en.wikipedia.org/wiki/OSI_model#cite_note-12) [SSH](https://en.wikipedia.org/wiki/Secure_Shell),[[13]](https://en.wikipedia.org/wiki/OSI_model#cite_note-13) [Telnet](https://en.wikipedia.org/wiki/Telnet)[[14]](https://en.wikipedia.org/wiki/OSI_model#cite_note-14) |
| 4. [Transport](https://en.wikipedia.org/wiki/Transport_layer) | [Segment](https://en.wikipedia.org/wiki/Packet_segmentation) (TCP) / [Datagram](https://en.wikipedia.org/wiki/Datagram) (UDP) | Reliable transmission of data segments between points on a network, including [segmentation](https://en.wikipedia.org/wiki/Packet_segmentation), [acknowledgement](https://en.wikipedia.org/wiki/Acknowledgement_(data_networks)) and [multiplexing](https://en.wikipedia.org/wiki/Multiplexing) | [NBF](https://en.wikipedia.org/wiki/NetBIOS_Frames_protocol), [TCP](https://en.wikipedia.org/wiki/Transmission_Control_Protocol), [UDP](https://en.wikipedia.org/wiki/User_Datagram_Protocol) |
| **Media layers** | 3. [Network](https://en.wikipedia.org/wiki/Network_layer) | [Packet](https://en.wikipedia.org/wiki/Network_packet) | Structuring and managing a multi-node network, including [addressing](https://en.wikipedia.org/wiki/Address_space),[routing](https://en.wikipedia.org/wiki/Routing) and [traffic control](https://en.wikipedia.org/wiki/Network_traffic_control) | [AppleTalk](https://en.wikipedia.org/wiki/AppleTalk), [ICMP](https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol), [IPsec](https://en.wikipedia.org/wiki/IPsec), [IPv4](https://en.wikipedia.org/wiki/IPv4), [IPv6](https://en.wikipedia.org/wiki/IPv6) |
| 2. [Data link](https://en.wikipedia.org/wiki/Data_link_layer) | [Frame](https://en.wikipedia.org/wiki/Frame_(networking)) | Reliable transmission of data frames between two nodes connected by a physical layer | [IEEE 802.2](https://en.wikipedia.org/wiki/IEEE_802.2), [L2TP](https://en.wikipedia.org/wiki/L2TP), [LLDP](https://en.wikipedia.org/wiki/LLDP), [MAC](https://en.wikipedia.org/wiki/Media_access_control), [PPP](https://en.wikipedia.org/wiki/Point-to-Point_Protocol), [ATM](https://en.wikipedia.org/wiki/Asynchronous_Transfer_Mode), [MPLS](https://en.wikipedia.org/wiki/Multiprotocol_Label_Switching) |
| 1. [Physical](https://en.wikipedia.org/wiki/Physical_layer) | [Bit](https://en.wikipedia.org/wiki/Bit) | Transmission and reception of raw bit streams over a physical medium | [DOCSIS](https://en.wikipedia.org/wiki/DOCSIS), [DSL](https://en.wikipedia.org/wiki/DSL), [Ethernet physical layer](https://en.wikipedia.org/wiki/Ethernet_physical_layer), [ISDN](https://en.wikipedia.org/wiki/ISDN), [RS-232](https://en.wikipedia.org/wiki/RS-232) |

See <https://en.wikipedia.org/wiki/OSI_model>





# 开发流程

## 敏捷开发

## 编码

静态检查工具

Coverity

cov\_pltf -saveview -savetmp -baseline ${BL\_PVOB}

+ cov\_pltf -saveview -savetmp -baseline B\_ENODEB\_LB\_COMP\_PLTF161CRMETROFDD8\_LR161\_160519.1@/enba\_pvob

.. (02:37:51) checking environment (raven.mh.lucent.com).

==> ... Coverity SA Version: 7.0.3

==> ... Running in Commit Parallel Build Mode for Stream: pltf\_lr161\_cr\_metro\_fdd\_integration1@/enba\_pvob

==> ... Coverity CIM:pltf DB:pltfcovmh

==> ... Using Coverity Compiler Config: /opt/swe/tools/in/projects/ltecovconf-1.0/prevent-7.0.3/coverity\_config.xml

==> creating RO Coverity view: loadbdr-coverity-20160519023756

.. (02:38:34/00:00:43) starting build 160519.0 for project pltf\_lr161\_cr\_metro\_fdd\_integration1

==> running upda.

.. (02:44:29/00:00:00) start cov\_build eccm2 (parallel build)

.. (02:44:29/00:00:00) start cov\_build eccm (parallel build)

.. (02:44:29/00:00:00) start cov\_build bcem (parallel build)

.. (02:44:29/00:00:00) start cov\_build host (parallel build)

.. (02:44:29/00:00:00) start cov\_build metroP3041 (parallel build)

.. (02:44:29/00:00:00) start cov\_build lrsocmetroB4860 (parallel build)

.. (02:44:29/00:00:00) start cov\_build lrhral (parallel build)

.. (02:44:29/00:00:00) start cov\_build lrsoc (parallel build - 5 minute delay)

==> cov\_build lrhral 385 compiles; 100% analyzed.

==> cov\_build bcem 1269 compiles; 100% analyzed.

==> cov\_build host 726 compiles; 100% analyzed.

==> cov\_build eccm 1344 compiles; 100% analyzed.

==> cov\_build metroP3041 1342 compiles; 100% analyzed.

==> cov\_build lrsocmetroB4860 1847 compiles; 100% analyzed.

==> cov\_build eccm2 1635 compiles; 100% analyzed.

==> cov\_build lrsoc 2165 compiles; 100% analyzed.

.. (03:41:49/00:57:20) waiting for Coverity builds and analyses.

.. (03:41:49/00:57:20) completed analyze for pltf\_lr161\_cr\_metro\_fdd\_integration1-bcem.

.. (03:41:49/00:57:20) completed analyze for pltf\_lr161\_cr\_metro\_fdd\_integration1-host.

.. (03:41:49/00:57:20) completed analyze for pltf\_lr161\_cr\_metro\_fdd\_integration1-lrhral.

.. (03:44:48/01:00:19) completed analyze for pltf\_lr161\_cr\_metro\_fdd\_integration1-metroP3041.

.. (03:46:07/01:01:38) completed analyze for pltf\_lr161\_cr\_metro\_fdd\_integration1-eccm.

.. (03:49:56/01:05:27) completed analyze for pltf\_lr161\_cr\_metro\_fdd\_integration1-lrsocmetroB4860.

.. (03:59:53/01:15:24) completed analyze for pltf\_lr161\_cr\_metro\_fdd\_integration1-lrsoc.

.. (04:00:05/01:15:36) completed analyze for pltf\_lr161\_cr\_metro\_fdd\_integration1-eccm2.

.. (04:00:12/01:15:43) starting Coverity commits.

.. (04:00:12) start cov\_commit lrsocmetroB4860.

.. (04:26:25) start cov\_commit lrsoc.

.. (04:30:09) start cov\_commit lrhral.

.. (04:30:47) start cov\_commit eccm2.

.. (04:40:20) start cov\_commit eccm.

.. (04:42:48) start cov\_commit bcem.

.. (04:45:16) start cov\_commit host.

.. (04:46:37) start cov\_commit metroP3041.

==> NOTICE: NO outstanding CIDs that were first detected on this UCM stream.

.. (04:49:01/02:11:10) completed successfully.

==> view loadbdr-coverity-20160519023756 is saved.

==> temporary directory saved at /prod/whsde/stc/covBU/tmp/cov\_pltf\_loadbdr\_62420160519023751

==> logfile saved at /prod/whsde/stc/covBU/tmp/cov\_pltf\_loadbdr\_62420160519023751/cumlog

exit $?+ exit 0

## Code Review

Collaborator

[https://collab.web.alcatel-lucent.com:8181/ui#](https://collab.web.alcatel-lucent.com:8181/ui)

## 单元测试（UT）

## 集成测试（CI）

## 系统测试（FI、ST）

## 代码提交（deliver）

Clearcase/DCT