linux信号-应用编程-专题讲座

written by 王保明

# linux信号基本概念

## 1基本概念

|  |
| --- |
| 中断   * 中断是系统对于异步事件的响应（） * 中断信号 * 中断源 * 现场信息 * 中断处理程序 * 中断向量表   异步事件的响应：进程执行代码的过程中可以随时被打断，然后去执行异常处理程序  生活中的中断和计算机系统中的中断   1. 无中断生活场景   张三看书，厨房烧水  2）有中断的生活场景  张三看书，设置闹钟，厨房烧水。  闹钟发出中断信号，张三把书合好(第20页)，去厨房把开水事情处理好，张三重新打开20页进行阅读。  3）计算机系统的中断场景  中断源发出中断信号，CPU判断中断是否屏蔽屏蔽、保护现场 ，cpu执行中断处理程序， cpu恢复现场，继续原来的任务。 4）中断的其他概念  中断向量表保存了中断处理程序的入口地址。  中断个数固定，操作系统启动时初始化中断向量表。  中断有优先级（有人敲门，有人打电话，有优先级）  中断可以屏蔽（张三可以屏蔽电话）。 |
| 中断分类   * 硬件中断（外部中断）   外部中断是指由外部设备通过硬件请求的方式产生的中断，也称为硬件中断   * 软件中断（内部中断）   内部中断是由CPU运行程序错误或执行内部程序调用引起的一种中断，也称为软件中断。   * x86平台INT指令 ARM软中断指令SWI |
| 信号概念 |
| * 信号是UNIX系统响应某些状况而产生的事件，进程在接收到信号时会采取相应的行动。 * 信号是因为某些错误条件而产生的，比如内存段冲突、浮点处理器错误或者非法指令等 * 信号是在软件层次上对中断的一种模拟，所以通常把它称为是软中断 |
| 信号和中断的区别   * 信号与中断的相似点：   + （1）采用了相同的异步通信方式；   + （2）当检测出有信号或中断请求时，都暂停正在执行的程序而转去执行相应的处理程序；   + （3）都在处理完毕后返回到原来的断点；   + （4）对信号或中断都可进行屏蔽。 * 信号与中断的区别：   + （1）中断有优先级，而信号没有优先级，所有的信号都是平等的；   + （2）信号处理程序是在用户态下运行的，而中断处理程序是在核心态下运行；   + （3）中断响应是及时的，而信号响应通常都有较大的时间延迟。 |

## 2信号名称及常用信号

|  |
| --- |
| 信号名称 描述  SIGABRT 进程停止运行 6  SIGALRM 警告钟  SIGFPE 算述运算例外  SIGHUP 系统挂断  SIGILL 非法指令  SIGINT 终端中断 2  SIGKILL 停止进程(此信号不能被忽略或捕获)  SIGPIPE 向没有读者的管道写入数据  SIGSEGV 无效内存段访问  SIGQUIT 终端退出 3  SIGTERM 终止  SIGUSR1 用户定义信号1  SIGUSR2 用户定义信号2  SIGCHLD 子进程已经停止或退出  SIGCONT 如果被停止则继续执行  SIGSTOP 停止执行  SIGTSTP 终端停止信号  SIGTOUT 后台进程请求进行写操作  SIGTTIN 后台进程请求进行读操作 |
| 实验1：  kill –l 可以查看linux内核支持的信号  Man 7 signal 查看信号的默认动作、信号的含义 |

## 3信号处理

|  |
| --- |
| 进程对信号的三种相应   * 忽略信号   不采取任何操作、有两个信号不能被忽略：SIGKILL（9号信号）和SIGSTOP。  思考1：为什么进程不能忽略SIGKILL、SIGSTOP信号。（如果应用程序可以忽略这2个信号，系统管理无法杀死、暂停进程，无法对系统进行管理。）。SIGKILL（9号信号）和SIGSTOP信号是不能被捕获的。   * 捕获并处理信号   内核中断正在执行的代码，转去执行先前注册过的处理程序。   * 执行默认操作   默认操作通常是终止进程，这取决于被发送的信号。  信号的默认操作：通过 man 7 signal 进程查看 |
| man 7 signal |
| 编程实践:让应用程序捕捉ctrl+c信号  以ctrl+c 会产生一个中断。当前应用程序捕捉ctrl+c中断信号。 |

## 4信号signal函数编程实践

signal信号安装函数

|  |
| --- |
| signal函数，作用1：站在应用程序的角度，注册一个信号处理函数。  作用2：忽略信号、设置信号默认处理 信号的安装和恢复   * typedef void (\*\_\_sighandler\_t) (int); * #define SIG\_ERR ((\_\_sighandler\_t) -1) * #define SIG\_DFL ((\_\_sighandler\_t) 0) * #define SIG\_IGN ((\_\_sighandler\_t) 1) * 函数原型:   **\_\_sighandler\_t signal(int signum, \_\_sighandler\_t handler);**   * **参数**   + signal是一个带signum和handler两个参数的函数，准备捕捉或屏蔽的信号由参数signum给出，接收到指定信号时将要调用的函数由handler给出   + handler这个函数必须有一个int类型的参数（即接收到的信号代码），它本身的类型是void   + handler也可以是下面两个特殊值：   SIG\_IGN 屏蔽该信号  SIG\_DFL 恢复默认行为 |
| 编程实践 注册SIGINT、注册SIGQUIT  编程实践：信号的安装和恢复  //测试信号的安装与恢复  int main(void)  {  \_\_sighandler\_t oldHandle;  //sighandler\_t \*oldHandle;  printf("main ....begin\n");  oldHandle = signal(SIGINT, myhandle);  if (oldHandle == SIG\_ERR)  {  perror("func signal err\n");  return 0;  }    printf("if u enter a, reset signal \n");  while(getchar() != 'a')  {  ;  }  //键入a以后，恢复默认函数  /\*  if (signal(SIGINT, oldHandle) == SIG\_ERR)  {  perror("func signal err\n");  return 0;  }  \*/    if (signal(SIGINT, SIG\_DFL) == SIG\_ERR)  {  perror("func signal err\n");  return 0;  }    while(1) ;  return 0;  }注意：signal函数函数返回值  RETURN VALUE  The signal() function returns the previous value of the signal handler, or SIG\_ERR on error. |
|  |
|  |
|  |

## 5信号分类

|  |
| --- |
| 不可靠信号PK可靠信号   * linux信号机制基本上是从unix系统中继承过来的。早期unix系统中的信号机制比较简单和原始，后来在实践中暴露出一些问题，它的主要问题是：   + 进程每次处理信号后，就将对信号的响应设置为默认动作。在某些情况下，将导致对信号的错误处理；因此，用户如果不希望这样的操作，那么就要在信号处理函数结尾再一次调用signal()，重新安装该信号。   + 早期unix下的不可靠信号主要指的是进程可能对信号做出错误的反应以及信号可能丢失。   + linux支持不可靠信号，但是对不可靠信号机制做了改进：在调用完信号处理函数后，不必重新调用该信号的安装函数（信号安装函数是在可靠机制上的实现）。因此，linux下的不可靠信号问题主要指的是信号可能丢失。 |
| 总结：unix信号机制不可靠地方，1）处理完信号以后，需要重新再注册信号；2）信号可能丢失。linux下已经对1做了优化。 |
| 可靠信号   * 随着时间的发展，实践证明，有必要对信号的原始机制加以改进和扩充。所以，后来出现的各种unix版本分别在这方面进行了研究，力图实现"可靠信号"。由于原来定义的信号已有许多应用，不好再做改动，最终只好又新增加了一些信号，并在一开始就把它们定义为可靠信号，这些信号支持排队，不会丢失。同 时，信号的发送和安装也出现了新版本：信号发送函数sigqueue()及信号安装函数sigaction()。 |
| 总结：sigaction和signal函数，都是调用内核服务do\_signal函数；  内核服务函数（应用系统无法调用的函数） |
| 实时信号（都是可靠信号）和非实时信号（不可靠信号）   * 早期Unix系统只定义了32种信号，Ret hat7.2支持64种信号，编号0-63(SIGRTMIN=31，SIGRTMAX=63)，将来可能进一步增加，这需要得到内核的支持。前32种信号已经有了预定义值，每个信号有了确定的用途及含义，并且每种信号都有各自的缺省动作。如按键盘的CTRL+C时，会产生SIGINT信号，对该信号的默认反应就是进程终止。后32个信号表示实时信号，等同于前面阐述的可靠信号。这保证了发送的多个实时信号都被接收。实时信号是POSIX标准的一部分，可用于应用进程。 * 非实时信号都不支持排队，都是不可靠信号；实时信号都支持排队，都是可靠信号。 |
| 实时信号和非实时信号   Man 7 signal 查看手册  非实时信号都不支持排队，都是不可靠信号；实时信号都支持排队，都是可靠信号 |

## 6信号发送

### kill函数

|  |
| --- |
| Kill基本用法 |
| 发送信号的函数有kill和raise  区别：kill既可以向自身发送信号，也可以向其他进程发送信号；  raise函数向进程自身发送信号。  Int kill(pid\_t pid, int siq)  int raise(int signo) |
| Int kill(pid\_t pid, int siq)  参数组合情况解释：  kill(pid\_t pid, int siq)  pid>0 将信号sig发给pid进程  pid=0 将信号sig发给同组进程  pid=-1 将信号sig发送给所有进程，调用者进程有权限发送的每一个进程（除了1号进程之外，还有它自身）  pid<-1 将信号sig发送给进程组是pid（绝对值）的每一个进程 |
| 实验：  //子进程向父进程发送信号  //子进程向同组进程发送信号 （getpgrp()函数获取进程组pid）  结论：注意，如果在fork之前安装信号，则子进程可以继承信号。 |
| kill和sleep在一起 |
| 子进程向父进程发送信号  sleep函数几点说明  1）sleep函数作用，让进程睡眠。  2）能被信号打断，然后处理信号函数以后，就不再睡眠了。直接向下执行代码  3）sleep函数的返回值，是剩余的秒数 |
|  |

### raise函数

|  |
| --- |
| raise   * raise   + 给自己发送信号。raise(sig)等价于kill(getpid(), sig); * killpg   + 给进程组发送信号。killpg(pgrp, sig)等价于kill(-pgrp, sig); * sigqueue   + 给进程发送信号，支持排队，可以附带信息。 |

### pause函数

|  |
| --- |
| pause()函数   * 将进程置为可中断睡眠状态。然后它调用内核函数schedule()，使linux进程调度器找到另一个进程来运行。 * pause使调用者进程挂起，直到一个信号被捕获 |

### alarm函数

|  |
| --- |
| alarm函数，设置一个闹钟延迟发送信号  告诉linux内核n秒中以后，发送SIGALRM信号；；  手册描述：  NAME  alarm - set an alarm clock for delivery of a signal  SYNOPSIS  #include <unistd.h>  unsigned int alarm(unsigned int seconds);  DESCRIPTION  alarm() arranges for a SIGALRM signal to be delivered to the process in seconds seconds.  If seconds is zero, no new alarm() is scheduled.  In any event any previously set alarm() is cancelled. |
| 实验1：手工发送信号alarm信号实践  kill -ALRM 21333  kill -14 213333  kill -SIGALRM 213333  命令发送信号实践  kill -alram ` ps -aux | grep 01aram | grep -v vi | awk '{print $2}' ` |
| 编程实践  void myhandle(int num)  {  printf("recv signal id num : %d \n", num);  //kill -alram ` ps -aux | grep 01aram | grep -v vi | awk '{print $2}' `  alarm(1);  }  int main(void)  {  printf("main ....begin\n");  //注册信号处理函数  if (signal(SIGALRM, myhandle) == SIG\_ERR)  {  perror("func signal err\n");  return 0;  }    alarm(1);  while(1)  {  pause();  printf("pause return\n");  }  return 0;  } |

## 8信号处理函数遇上可重入和不可重入函数

|  |
| --- |
| 可重入函数概念   * 为了增强程序的稳定性，在信号处理函数中应使用可重入函数。 * 所谓可重入函数是指一个可以被多个任务调用的过程，任务在调用时不必担心数据是否会出错。因为进程在收到信号后，就将跳转到信号处理函数去接着执行。如果信号处理函数中使用了不可重入函数，那么信号处理函数可能会修改原来进程中不应该被修改的数据，这样进程从信号处理函数中返回接着执行时，可能会出现不可预料的后果。不可再入函数在信号处理函数中被视为不安全函数。 * 满足下列条件的函数多数是不可再入的：（1）使用静态的数据结构，如getlogin()，gmtime()，getgrgid()，getgrnam()，getpwuid()以及getpwnam()等等；（2）函数实现时，调用了malloc（）或者free()函数；（3）实现时使用了标准I/O函数的 |
| man手册实践  man 7 signal 查找可重入函数和不可重入函数，下一个内核用文件描述符来优化信号 |
| 结论：  在信号处理函数中，尽量不使用全局变量和静态变量的函数。特别是这个变量在程序中随时读写。 |

## 9信号的阻塞和未达

### 信号在linux中的传递过程

|  |
| --- |
| 信号在内核中的表示   * 执行信号的处理动作称为信号*递达*（Delivery），信号从产生到递达之间的状态，称为信号*未决*（Pending）。进程可以选择阻塞（Block）某个信号。被阻塞的信号产生时将保持在未决状态，直到进程解除对此信号的阻塞，才执行递达的动作。 * 注意，阻塞和忽略是不同，只要信号被阻塞就不会递达，而忽略是在递达之后可选的一种处理动作。信号在内核中的表示可以看作是这样的： |
| 说明1）PCB进程控制块中函数有信号屏蔽状态字（block）信号未决状态字（pending）还有是否忽略标志；  说明2）信号屏蔽状态字（block），1代表阻塞、0代表不阻塞；信号未决状态字（pending）的1代表未决，0代表信号可以抵达了；  说明3）向进程发送SIGINT，内核首先判断信号屏蔽状态字是否阻塞，信号未决状态字（pending相应位制成1；若阻塞解除，信号未决状态字（pending）相应位制成0；表示信号可以抵达了。  说明4）block状态字、pending状态字 64bit；  说明5）block状态字用户可以读写，pending状态字用户只能读；这是信号设计机制。  思考1：状态字都64bit，编程时，如何表示状态字那？  思考2：block状态字信息如何获取或者操作那？哪些api？  思考3：pending状态字信息如何获取或者操作那？哪些api？ |
| 信号集操作函数（状态字表示）   * #include <signal.h> * int sigemptyset(sigset\_t \*set); 把信号集情况64bit/8=8个字节 * int sigfillset(sigset\_t \*set); 把信号集情况1 * int sigaddset(sigset\_t \*set, int signo); 根据signo，把信号集中的对应为置成1 * int sigdelset(sigset\_t \*set, int signo); 根据signo，把信号集中的对应为置成0 * int sigismember(const sigset\_t \*set, int signo);//判断signo是否在信号集中 |
| sigprocmask读取或更改进程的信号屏蔽状态字（block）   * #include <signal.h> * int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oset); * 功能：读取或更改进程的信号屏蔽字。 * 返回值：若成功则为0，若出错则为-1 * 如果oset是非空指针，则读取进程的当前信号屏蔽字通过oset参数传出。如果set是非空指针，则更改进程的信号屏蔽字，参数how指示如何更改。如果oset和set都是非空指针，则先将原来的信号屏蔽字备份到oset里，然后根据set和how参数更改信号屏蔽字。假设当前的信号屏蔽字为mask，下表说明了how参数的可选值。   how含义    说明：SIG\_BLOCK ，讲信号集set添加到进程block状态字中。 |
| sigpending获取信号未决状态字（pending）信息  NAME  sigpending - examine pending signals  SYNOPSIS  #include <signal.h>  int sigpending(sigset\_t \*set);  DESCRIPTION  sigpending() returns the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). The mask of pending signals is returned in set. |

### 信号阻塞未达编程实践

|  |
| --- |
| 综合实验，  SIGINT信号未设置阻塞，查看未决关键字  SIGINT信号设置阻塞，查看未决关键字  SIGINT信号解除阻塞，查看未决关键字（通过ctrl+\ SIGQUIT 解除阻塞）  //演示信号从产生到抵达的整个过程  //信号的阻塞和解除阻塞综合实验  //设置信号阻塞和非阻塞，设置ctl+q来解除信号阻塞  void handler(int sig)  {  if (sig == SIGINT)  printf("recv a sig=%d\n", sig);  else if (sig == SIGQUIT)  {  sigset\_t uset;  sigemptyset(&uset);  sigaddset(&uset, SIGINT);  //ctr + \ 用来接触 SIGINT 信号  //解除阻塞  sigprocmask(SIG\_UNBLOCK, &uset, NULL);  }  }  void printsigset(sigset\_t \*set)  {  int i;  for (i=1; i<NSIG; ++i)  {  if (sigismember(set, i))  putchar('1');  else  putchar('0');  }  printf("\n");  }  //3 连续的按ctrl+c键盘，虽然发送了多个SIGINT信号，但是因为信号是不稳定的，只保留了一个。  //不支持排队  int main(int argc, char \*argv[])  {  sigset\_t pset; //用来打印的信号集  sigset\_t bset; //用来设置阻塞的信号集    sigemptyset(&bset);  sigaddset(&bset, SIGINT);    if (signal(SIGINT, handler) == SIG\_ERR)  ERR\_EXIT("signal error");    if (signal(SIGQUIT, handler) == SIG\_ERR)  ERR\_EXIT("signal error");  //读取或更改进程的信号屏蔽字 这里用来阻塞ctrl+c信号  //ctrl+c信号被设置成阻塞，即使用户按下ctl+c键盘，也不会抵达  sigprocmask(SIG\_BLOCK, &bset, NULL);    for (;;)  {  //获取未决 字信息  sigpending(&pset);    //打印信号未决 sigset\_t字  printsigset(&pset);  sleep(1);  }  return 0;  } |
| 杀死进程，命令行 |
| [wbm01@localhost ~]$ ps -u wbm01 | grep "dm01" | awk '{print $1}'  1496  [wbm01@localhost ~]$ kill -9 `ps -u wbm01 | grep "dm01" | awk '{print $1}' ` |

### 从linux信号机制编程实践说开去

没有看不懂的代码，只有看不懂的业务

学习linux应用编程、系统编程方法谈，重要的是

1：对各种linux各个机制搭建、深入理解；

2：思想兑现代码！（加强实践，多动手，加强考试！）===》逼迫自己强制使用！

## 10信号的高级用法

### sigaction函数注册信号处理函数

|  |
| --- |
| **sigaction函数**   * 包含头文件<signal.h> * 功能:sigaction函数用于改变进程接收到特定信号后的行为。 * 原型：   **int sigaction(int signum,const struct sigaction \*act,const struct sigaction \*old);**   * 参数   + 该函数的第一个参数为信号的值，可以为除SIGKILL及SIGSTOP外的任何一 个特定有效的信号（为这两个信号定义自己的处理函数，将导致信号安装错误）   + 第二个参数是指向结构sigaction的一个实例的指针，在结构 sigaction的实例中，指定了对特定信号的处理，可以为空，进程会以缺省方式对信号处理   + 第三个参数oldact指向的对象用来保存原来对相应信号的处理，可指定oldact为NULL。 * 返回值:函数成功返回0，失败返回-1   signal(num., handle) |
| **sigaction结构体**   * 第二个参数最为重要，其中包含了对指定信号的处理、信号所传递的信息、信号处理函数执行过程中应屏蔽掉哪些函数等等   struct sigaction {  void (\*sa\_handler)(int); //信号处理程序 不接受额外数据  void (\*sa\_sigaction)(int, siginfo\_t \*, void \*); //信号处理程序 能接受额外数据，和sigqueue配合使用  sigset\_t sa\_mask; //  int sa\_flags; //影响信号的行为 SA\_SIGINFO表示能接受数据  void (\*sa\_restorer)(void); //废弃  };  注意1：回调函数句柄sa\_handler、sa\_sigaction只能任选其一。  注意2：The sigaction structure is defined as something like 思考如何测试？ |
| **会查找、会用man手册，是通往高手的必经之路。**  The siginfo\_t parameter to sa\_sigaction is a struct with the following elements  siginfo\_t {  int si\_signo; /\* Signal number \*/  int si\_errno; /\* An errno value \*/  int si\_code; /\* Signal code \*/  pid\_t si\_pid; /\* Sending process ID \*/  uid\_t si\_uid; /\* Real user ID of sending process \*/  int si\_status; /\* Exit value or signal \*/  clock\_t si\_utime; /\* User time consumed \*/  clock\_t si\_stime; /\* System time consumed \*/  sigval\_t si\_value; /\* Signal value \*/  int si\_int; /\* POSIX.1b signal \*/  void \* si\_ptr; /\* POSIX.1b signal \*/  void \* si\_addr; /\* Memory location which caused fault \*/  int si\_band; /\* Band event \*/  int si\_fd; /\* File descriptor \*/  } |
| **实验1：**sigaction的函数注册信号，基本用法  void handler(int sig)  {  printf("recv a sig=%d\n", sig);  }  \_\_sighandler\_t my\_signal(int sig, \_\_sighandler\_t handler)  {  struct sigaction act;  struct sigaction oldact;  act.sa\_handler = handler;  sigemptyset(&act.sa\_mask);  act.sa\_flags = 0;  if (sigaction(sig, &act, &oldact) < 0)  return SIG\_ERR;  return oldact.sa\_handler;  }  int main(int argc, char \*argv[])  {  struct sigaction act;  sigset\_t sa\_mask;    act.sa\_handler = handler;  act.sa\_flags = 0;  sigemptyset(&act.sa\_mask);  //测试信号安装函数  //sigaction(SIGINT, &act, NULL);    //模拟signal函数  my\_signal(SIGINT, handler);  for (;;)  {  pause();  }  return 0;  } |
| 实验2：测试sigaction结构体第三个参数sigset\_t sa\_mask的作用  /\*  struct sigaction {  void (\*sa\_handler)(int);  void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);  sigset\_t sa\_mask;  int sa\_flags;  void (\*sa\_restorer)(void);  } \*/  //测试sigaction结构体第三个参数sigset\_t sa\_mask的作用  //作用 sigaddset(&act.sa\_mask, SIGQUIT); 加入到sa\_mask中的信号，被阻塞（信号处理函数执行的过程中被阻塞）。  //注意：SIGQUIT信号最终还会抵达  int main(int argc, char \*argv[])  {  struct sigaction act;  act.sa\_handler = handler;    sigemptyset(&act.sa\_mask);  sigaddset(&act.sa\_mask, SIGQUIT);  act.sa\_flags = 0;  if (sigaction(SIGINT, &act, NULL) < 0)  ERR\_EXIT("sigaction error");  for (;;)  pause();  return 0;  }  void handler(int sig)  {  printf("recv a sig=%d 信号处理函数执行的时候，阻塞sa\_mask中的信号\n", sig);  sleep(5);  } |

### sigqueue新的信号发送函数

|  |
| --- |
| **sigqueue函数**   * 功能:新的发送信号系统调用,主要是针对实时信号提出的支持信号带有参数,与函数sigaction()配合使用。 * 注意：和kill函数相比Int kill(pid\_t pid, int siq)多了参数 * 原型:   **int sigqueue(pid\_t pid, int sig, const union sigval value);**   * 参数   + sigqueue的第1个参数是指定接收信号的进程id，第2个参数确定即将发送的信号，第3个参数是一个联合数据结构union sigval，指定了信号传递的参数，即通常所说的4字节值。 * 返回值成功返回0,失败返回-1 |
| * 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。 * sigval联合体   typedef union sigval  {  int sival\_int;  void \*sival\_ptr;  }sigval\_t; |

## 12man手册的使用



