# linux进程专题讲座

## 从linux角度看linux应用编程

|  |
| --- |
|  |
| 操作系统三个基本抽象   * 操作系统通过三个基本抽象概念来实现这两个功能。（进程、虚拟存储器和文件）。文件是对I/O设备的抽象表示，虚拟存储器是对主存和磁盘I/O设备的抽象表示，进程则是对处理器、主存和I/O设备的抽象表示。 |
|  |
|  |
|  |
|  |

## 基本概念

|  |
| --- |
| 进程三态 |
| Linux内核中的进程状态   * 运行状态（TASK\_RUNNING） * 可中断睡眠状态（TASK\_INTERRUPTIBLE） * 不可中断睡眠状态（TASK\_UNINTERRUPTIBLE） * 暂停状态（TASK\_STOPPED） * 僵死状态（TASK\_ZOMBIE） |
| 进程的相关理解   * 理解1：fork系统调用之后，父子进程将交替执行。 * 理解2：怎么样理解一次调用2次返回? * 理解3：怎么样理解，fork返回值大于零的是父进程，为什么这样设计:？   理解4：怎么样理解分支在fork之后，而不是父进程main函数的开始？ |
| 孤儿进程和僵尸进程  产生原因和避免方法 |
| fork之后父子进程共享文件 |
|  |
|  |
|  |

## fork父子进程遇上共享文件

|  |
| --- |
| 1：fork子进程拷贝父进程的数据段  Vfork子进程与父进程共享数据段；  2：fork父、子进程的执行次序不确定  Vfork：子进程先运行，父进程后运行； |
| Vfork和exec函数族在一起  execve替换进程映像（加载程序）注意execve是一个系统调用;  替换意味着：代码段、数据段、堆栈段、进程控制块PCB全部替换。 |

## 进程终止的5种方式

|  |
| --- |
| 进程终止的5种方式   * 正常退出   + 从main函数返回   + 调用exit   + 调用\_exit * 异常退出   + 调用abort 产生SIGABOUT信号   + 由信号终止 **ctrl+c SIGINT** |
| void bye1(void)  {  printf("bye1...do\n");  }  void bye2(void)  {  printf("bye2...do\n");  }    int main01()  {  atexit(bye1);    atexit(bye2);  printf("hello.... fddddddddd1111");  fflush(stdout);    //exit(0) //是c库函数 =======其他操作。。。。在调用\_exit  //\_exit(0); //系统调用  } |

## exec函数族替换进程印象

|  |
| --- |
| extern char \*\*environ;  int execl(const char \*path, const char \*arg, ...);  int execlp(const char \*file, const char \*arg, ...); PATH  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[]); |
|  |
| int main(void )  {  pid\_t pid;    printf("getpid():%d\n", getpid());  //int execlp(const char \*file, const char \*arg, ...);  // int execle(const char \*path, const char \*arg,  // ..., char \* const envp[]);    char \* const envp[] = {"aaa=111", "bbb=2222", NULL};    // int execle(const char \*path, const char \*arg,  // ..., char \* const envp[]);  //当你不填写envp时，默认传递当前用户的env环境变量、到hello印象中  //当你填写envp时，传递你写的envp  //env shell的环境变量的概念  execle("./hello", "aaa", "bbb", "ccc", NULL, NULL);  //要知道extern char \*\*environ;是谁分配的内存  printf("hello\n");  return 0;  } |
| hello应用程序  extern char \*\*environ;  int main()  {  int i = 0;  printf("getpid():%d\n", getpid());    for (i=0; environ[i] != NULL; i++)  {  printf("%s\n", environ[i]);  }    return 0;    } |

一个应用把另外一个应用拉起来

函数组中的环境变量

函数族学习技巧

6、父进程wait函数waitpid

技术出现原因 wait和waitpid函数用法

WIFEXITED WEXITSTATUS WIFSIGNALED WTERMSIG WIFSTOPPED WSTOPSIG

7、system C库函数

系统函数使用及自动动手写system函数

8、守护进程

控制终端、会话期、进程组

守护进程思想及编程实践（系统函数、自动动手写）

文件描述符及文件描述符的赋值

进程项目开发进价

多进程测试框架

多进程遇上动态库

# linux信号专题讲座

## 信号的基本概念

|  |
| --- |
| 1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  2异步通知事件。。。。。本质：顺序执行的代码可以被信号机制打断。。  3信号与中断的区别  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 |
|  |
|  |

## 信号的安装和恢复

|  |
| --- |
| 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;  } |

## 信号分类

不可靠信号PK可靠信号

## 信号发送函数

kill函数

|  |
| --- |
| 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 |
| //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） |
| 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，可以附带额外数据信息 |
| 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，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；；  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;  } |

## 可中断睡眠和不可中断睡眠

|  |
| --- |
| int main(void)  {  int n = 100;  pid\_t pid;  printf("main ....begin\n");  //SIGINT 是ctrl+c 会产生2号信号。。。 中断应用程序  signal(SIGINT, handler);    do  {  n = sleep(n); //sleep是可中断睡眠，让进程睡够  printf("要给睡够。。。。。\n");  } while(n > 0);      //wait();  printf("sleep ....结束");  } |

## 信号处理函数遇上可重入和不可重入函数

## 信号的阻塞和未达

|  |
| --- |
|  |
| 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bit置成1 * int sigdelset(sigset\_t \*set, int signo); 根据signo，把信号集中的对应bit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。 |
| 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;  } |
|  |

## 信号的高级用法 （重点难点）

|  |  |
| --- | --- |
| **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; |  |

|  |
| --- |
| void myhandle(int num, siginfo\_t \*info , void \* p)  {  if (num == SIGUSR1)  {  //把SIGINT和SIGRTMIN均添加到本进程的阻塞状态字中  sigset\_t bset;  sigemptyset(&bset);  sigaddset(&bset, SIGINT);  sigaddset(&bset, SIGRTMIN);    sigprocmask(SIG\_UNBLOCK, &bset, NULL);  printf("解除阻塞 recv sig num:%d \n", num);    }  else if (num == SIGINT || num== SIGRTMIN)  {  //printf("recv sig num:%d \n", num);  printf("收到的数据是，%d\n",info->si\_value.sival\_int);  }  else  {  printf("其他recv sig num:%d \n", num);  printf("收到的数据是，%d\n",info->si\_value.sival\_int);  }  }  void myhandle22(int num)  {  if (num == SIGUSR1)  {  //把SIGINT和SIGRTMIN均添加到本进程的阻塞状态字中  sigset\_t bset;  sigemptyset(&bset);  sigaddset(&bset, SIGINT);  sigaddset(&bset, SIGRTMIN);    sigprocmask(SIG\_UNBLOCK, &bset, NULL);  printf("解除阻塞 recv sig num:%d \n", num);  }  else if (num == SIGINT || num== SIGRTMIN)  {  printf("recv sig num:%d \n", num);  }  else  {  printf("其他recv sig num:%d \n", num);  }  }  void main()  {  pid\_t pid;    struct sigaction act;  act.sa\_handler = myhandle;  act.sa\_flags = SA\_SIGINFO; //千万别忘记了  //注册非实时信号 处理函数  if ( sigaction(SIGINT, &act, NULL) <0 )  {  ERR\_EXIT("sigaction SIGINT");  }  //注册实时信号的处理函数  if ( sigaction(SIGRTMIN, &act, NULL) <0 )  {  ERR\_EXIT("sigaction SIGINT");  }    //注册了一个用户自定义信号SIGUSR1 处理函数  if ( sigaction(SIGUSR1, &act, NULL) <0 )  {  ERR\_EXIT("sigaction SIGINT");  }    //把SIGINT和SIGRTMIN均添加到本进程的阻塞状态字中  sigset\_t bset;  sigemptyset(&bset);  sigaddset(&bset, SIGINT);  sigaddset(&bset, SIGRTMIN);    sigprocmask(SIG\_BLOCK, &bset, NULL);    pid = fork();  if (pid == -1)  {  ERR\_EXIT("fork err");  }    if (pid == 0)  {  int i = 0;  int ret = 0;  union sigval v;  v.sival\_int = 201;  /\*  //发三次不稳定信号  for (i=0; i<3\*1024; i++)  {  v.sival\_int ++;  ret = sigqueue(getppid(), SIGINT, v);  if (ret != 0)  {  printf("发送不可靠信号失败 ret: %d, errno:%d \n", ret, errno);  exit(0);  }  else  {  printf("发送不可靠信号ok\n");  }  }  \*/    v.sival\_int = 0;  //发三次稳定信号  for (i=0; i<9\*1024; i++) //8K  {  v.sival\_int ++;  ret = sigqueue(getppid(), SIGRTMIN, v);  if (ret != 0)  {  //printf("发送可靠信号失败 ret: %d, errno:%d \n", ret, errno);  //exit(0);  }  //printf("发送可靠信号ok i:%d\n", i);  }    sleep(2);  kill(getppid(), SIGUSR1);  }    while(1)  {  sleep(1);  }    printf("main....\n");  } |