

- 1.守护进程创建
- 1.1什么是守护进程
- 1.2守护进程创建的流程
- 1.3文件描述符重定向
- 1.3.1dup/dup2函数的功能
- 1.3.2dup文件描述符重定向实例
- 1.3.3dup2文件描述符重定向实例
- 1.4守护进程创建的实例
- 2.进程内程序替换函数
- 2.1system函数功能
- 2.2system函数实例
- 3.线程
- 3.1线程的概念
- 3.2线程的创建及特点
- 3.2.1多线程的创建
- 3.2.2线程创建的接口
- 3.2.3线程创建的实例1(不传参)
- 3.2.3线程创建的实例2（传参）
- 3.2.4线程创建的实例4
- 3.2.5线程创建的实例5
- 3.2.6多线程执行顺序
- 3.2.7多线程内存空间问题
- 3.3线程相关的函数
- 3.3.1pthread_self函数
- 3.3.2pthread_exit函数
- 3.3.3pthread_join函数
- 3.3.4多线程拷贝文件的练习
- 3.3.5pthread_detach函数
- 3.3.6pthread_cancel函数
- 3.4多线程访问全局变量问题

1.守护进程创建

1.1什么是守护进程

守护进程：守护进程是后台运行的进程，它会随着系统的启动而启动，

会随着系统的终止而终止，类似于windows上的各种服务。例如windows

上的网络管理的服务，ubuntu上的sshd的服务，ubuntu上的资源管理的进程

1.2守护进程创建的流程

1. 创建孤儿进程
2. 设置孤儿进程的会话id和组id

```
1 pid_t setsid(void);
2 功能：设置会话id
3 参数：
4     @无
5 返回值：成功返回新的会话id，失败返回-1，置位错误码
```

3. 切换目录到/var/log目录下

```
1 int chdir(const char *path);
2 功能：切换目录
3 参数：
4     @path:路径
5 返回值：成功返回0，失败返回-1置位错误码
```

4. 设置umask的值0

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3
4 mode_t umask(mode_t mask);
5 功能：设置文件的掩码
6 参数：
7     @mask:掩码值
8 返回值：返回设置前的掩码值，总是会成功
```

5. 创建日志文件

```
1 int fd;
2 if((fd = open("./daemon.log",O_RDWR|O_CREAT|O_APPEND,0666))== -1)
3     PRINT_ERR("open error");
```

6. 文件描述符重定向工作

```
1 dup2(fd,0);
2 dup2(fd,1);
3 dup2(fd,2);
```

7. 开启自己的服务

1.3文件描述符重定向

1.3.1dup/dup2函数的功能

```
1 #include <unistd.h>
2
3 int dup(int oldfd);
4 功能：将使用oldfd生成newfd,newfd采用文件描述符最小未使用的原则分配的。
5     oldfd和newfd都能操作文件，两者共用光标。
6 参数：
7     @oldfd:旧的文件描述符
8 返回值：成功返回新的文件描述符，失败返回-1置位错误码
9
10
11 int dup2(int oldfd, int newfd);
12 功能：将oldfd重定向到newfd中，newfd是用户自己指定的，如果newfd
13     之前被分配过在使用前会关闭它们。
14 参数：
15     @oldfd:旧的文件描述符
16     @newfd:新的文件描述符
17 返回值：成功返回新的文件描述符，失败返回-1置位错误码
```

1.3.2dup文件描述符重定向实例

```
1 #include <head.h>
2
3 int main(int argc, const char* argv[])
4 {
5     int fd;
6     if ((fd = open("./daemon.log", O_RDWR | O_CREAT | O_APPEND, 0666)) == -1)
7         PRINT_ERR("open error");
8
9     //关闭标准输入，标准输出，标准出错
10    close(0);
11    close(1);
12    close(2);
13
14    //将0,1,2重定向到文件fd中
15    dup(fd);
16    dup(fd);
17    dup(fd);
18
19    printf("hello daemon dup process...\n");
20    fflush(stdout);
21    fprintf(stderr,"this is test daemon dup stderr...\n");
22    lseek(fd,0,SEEK_SET);
23    char ch;
24    scanf("%c",&ch);
25    printf("ch = %c\n",ch);
26    return 0;
27 }
```

1.3.3dup2文件描述符重定向实例

```
1 #include <head.h>
2
3 int main(int argc, const char* argv[])
4 {
5     int fd;
6     if ((fd = open("./daemon.log", O_RDWR | O_CREAT | O_APPEND, 0666)) == -1)
7         PRINT_ERR("open error");
8
9     dup2(fd,0);
10    dup2(fd,1);
11    dup2(fd,2);
12
13    printf("hello daemon dup process...\n");
14    fflush(stdout);
15    fprintf(stderr,"this is test daemon dup stderr...\n");
16    lseek(fd,0,SEEK_SET);
17    char ch;
18    scanf("%c",&ch);
19    printf("ch = %c\n",ch);
20    return 0;
21 }
```

1.4守护进程创建的实例

```
1  #include <head.h>
2
3  int main(int argc, const char* argv[])
4  {
5      pid_t pid;
6      int fd;
7      if ((pid = fork()) == -1) {
8          PRINT_ERR("fork error");
9      } else if (pid == 0) {
10         // 1.孤儿进程
11         // 2.设置会话id
12         if(setsid()==-1)
13             PRINT_ERR("setsid error");
14         // 3.切换目录(/var/log/)
15         if (chdir("/"))
16             PRINT_ERR("chdir error");
17         // 4.修改掩码
18         umask(0);
19         // 5.创建日志文件
20         if ((fd = open("./daemon.log", O_RDWR | O_CREAT | O_APPEND, 0666)) == -1)
21             PRINT_ERR("open error");
22         // 6.文件描述符重定向
23         dup2(fd,0);
24         dup2(fd,1);
25         dup2(fd,2);
26         // 7.开启自己的服务
27         char s[] = "i am test daemon process\n";
28         while(1){
29             write(1,s,strlen(s));
30             sleep(1);
31         }
32
33     } else {
34         // 让父进程退出
35         exit(0);
36     }
37     return 0;
38 }
```

2.进程内程序替换函数

问：终端是如何执行a.out程序的？

答：终端进程首先会执行fork产生一个子进程，当产生子进程之后，子进程的.text

段存放的是终端的可执行程序，需要将这个段内的内容替换成a.out。此时就可以

执行a.out应用程序了，以上的功能可以通过system完成。

2.1system函数功能

```
1  #include <stdlib.h>
2
3  int system(const char *command);
4  功能：首先会fork一个子进程，在子进程内执行command这个可执行程序
5  参数：
6      @command:可执行程序的路径及名字
7  返回值：
8      *如果command为NULL，如果终端可用返回非0，如果终端不可用返回0
9      *如果无法创建子进程，或者无法检索其状态，则返回值为-1。
10     *如果命令不能在子进程中执行，返回退出状态（效果和exit(status)一样的）
11     *如果所有系统调用都成功，则返回值是用于执行命令的子shell的终止状态。
```

2.2system函数实例

```
1  #include <head.h>
2
3  int main(int argc, const char* argv[])
4  {
5      // 首先会创建一个子进程，在子进程内执行"/bin/ls"可执行程序
6      // if(system("/bin/ls -l"))
7      //     PRINT_ERR("system error");
8
9      // if(system("./b.out"))
10     //     PRINT_ERR("system error");
11
12     if (system("./myshell.sh 111 222 333 4444 555"))
13         PRINT_ERR("system error");
```

```
14
15         return 0;
16     }
```

3.线程

3.1线程的概念

线程（LWP）：线程是轻量级的进程，进程是分配资源的最小单位（0-3G）,线程是调度的最小单位。线程本身不占用资源它是共享进程的资源。线程没有进程安全，因为如果一个线程导致进程结束，其他所有的线程都不能执行。多线程的并发性比多进程的高，因为线程间切换比进程间切换时间短。线程间资源共享，所以线程间通信要比进程间通信更为容易。

```
1 ps -ajx ==>看进程附加态（1，代表多线程）
2 ps -eLf ==>多线程
3 htop      ==>多线程
```

3.2线程的创建及特点

3.2.1多线程的创建

多线程创建的接口是第三方库提供的libpthread.so，所以如果要使用线程相关的函数，在编译的时候就必须链接这个库gcc xxx.c -lpthread，线程相关的man手册需要使用apt-get来安装(sudo apt-get install manpages-*)。

3.2.2线程创建的接口

```
1 #include <pthread.h>
2
3 int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
4                    void *(*start_routine) (void *), void *arg);
5 功能：创建线程
6 参数：
7     @thread:线程号指针
8     @attr:线程属性（一般填写为NULL）
9     @start_routine: 线程体（指向线程处理函数）
10    @arg: 向线程体传递的参数（如果没有填写为NULL）
11 返回值：成功返回0，失败返回错误码
12 Compile and link with -pthread.
```

3.2.3线程创建的实例1(不传参)

```
1 #include <head.h>
2 void* task1(void* arg)
3 {
4     while (1) {
5         printf("我是子线程...\n");
6         sleep(1);
7     }
8 }
9 int main(int argc, const char* argv[])
10 {
11     pthread_t tid;
12
13     if ((errno = pthread_create(&tid, NULL, task1, NULL)) != 0)
14         PRINT_ERR("pthread_create error");
15
16     while (1) {
17         printf("我是主线程...\n");
18         sleep(1);
19     }
20     return 0;
21 }
```

3.2.3线程创建的实例2（传参）

```
1 #include <head.h>
2 typedef struct {
3     char name[20];
4     char sex;
5     int age;
6 }stu_t;
7 void* task1(void* arg)
8 {
9     stu_t *stu = (stu_t *)arg;
```

```
10 printf("name=%s,sex=%c,age=%d\n",stu->name,stu->sex,stu->age);
11 while (1) {
12     printf("我是子线程...\n");
13     sleep(1);
14 }
15 }
16 int main(int argc, const char* argv[])
17 {
18     pthread_t tid;
19     stu_t stu = {
20         .name = "zhangsan",
21         .sex = 'm',
22         .age = 20,
23     };
24
25     if ((errno = pthread_create(&tid, NULL, task1, (void *)&stu)) != 0)
26         PRINT_ERR("pthread_create error");
27
28     while (1) {
29         printf("我是主线程...\n");
30         sleep(1);
31     }
32     return 0;
33 }
```

3.2.4线程创建的实例4

如果定义一个全局变量，多线程都可以使用这个全局变量，因为共享内存空间

```
1 #include <head.h>
2 typedef struct {
3     char name[20];
4     char sex;
5     int age;
6 } stu_t;
7
8 stu_t stu = {
9     .name = "zhangsan",
10    .sex = 'm',
11    .age = 20,
12 };
13 void* task1(void* arg)
14 {
15     printf("child:name=%s,sex=%c,age=%d\n", stu.name, stu.sex, stu.age);
16     while (1) {
17         printf("我是子线程...\n");
18         sleep(1);
19     }
20 }
21 int main(int argc, const char* argv[])
22 {
23     pthread_t tid;
24     printf("parent:name=%s,sex=%c,age=%d\n", stu.name, stu.sex, stu.age);
25     if ((errno = pthread_create(&tid, NULL, task1,NULL))!=0)
26         PRINT_ERR("pthread_create error");
27
28     while (1) {
29         printf("我是主线程...\n");
30         sleep(1);
31     }
32     return 0;
33 }
```

3.2.5线程创建的实例5

```
1 #include <head.h>
2
3 void* task1(void* arg)
4 {
5     printf("我是子线程...\n");
6 }
7 int main(int argc, const char* argv[])
8 {
9     pthread_t tid;
10
11     if ((errno = pthread_create(&tid, NULL, task1, NULL)) != 0)
12         PRINT_ERR("pthread_create error");
13
14     printf("我是主线程...\n");
15
16     //如果不写这个sleep(1),有可能主线程执行结束，子线程还没来及执行
17     //进程就退出了，子线程将不能执行，所有为了让子线程正常执行，加了sleep(1);
18     sleep(1);
```

```
19 |         return 0;
20 |     }
```

3.2.6多线程执行顺序

多线程执行没有先后顺序，时间片轮询，上下文切换

3.2.7多线程内存空间问题

多线程共享进程的内存空间，全局变量多线程都可以访问，也都可以修改。

3.3线程相关的函数

3.3.1pthread_self函数

```
1 | #include <pthread.h>
2 | pthread_t pthread_self(void);
3 | 功能：获取当前线程的线程号
4 | 参数：
5 |     @无
6 | 返回值：总是会成功，返回线程号
```

```
1 | #include <head.h>
2 |
3 | void* task1(void* arg)
4 | {
5 |     printf("我是子线程...\n");
6 |     printf("子:tid = %ld\n",pthread_self());
7 | }
8 | int main(int argc, const char* argv[])
9 | {
10 |     pthread_t tid;
11 |
12 |     if ((errno = pthread_create(&tid, NULL, task1, NULL)) != 0)
13 |         PRINT_ERR("pthread_create error");
14 |
15 |     printf("主:tid = %ld,子:tid = %ld\n",pthread_self(),tid);
16 |
17 |     sleep(1);
18 |     return 0;
19 | }
```

3.3.2pthread_exit函数

在线程中一般不调用exit/_exit,因为这些函数会让进程退出，如果进程

退出了，所有的线程都将不能执行。所以如果想退出某个线程是用pthread_exit完成。

```
1 | #include <pthread.h>
2 | void pthread_exit(void *retval);
3 | 功能：退出线程
4 | 参数：
5 |     @retval:线程退出的状态
6 | 返回值：无
```

```
1 | #include <head.h>
2 |
3 | void* task1(void* arg)
4 | {
5 |     while(1){
6 |         sleep(1);
7 |     }
8 | }
9 | int main(int argc, const char* argv[])
10 | {
11 |     pthread_t tid;
12 |
13 |     if ((errno = pthread_create(&tid, NULL, task1, NULL)) != 0)
14 |         PRINT_ERR("pthread_create error");
15 |
16 |     printf("主:tid = %ld,子:tid = %ld\n", pthread_self(), tid);
17 |
18 |     int n = 5;
19 |     while (1) {
20 |         printf("我是主线程...\n");
21 |         sleep(1);
22 |         n--;
23 |         if (n == 0)
```

```
24 pthread_exit(NULL);
25 }
26 return 0;
27 }
```

3.3.3pthread_join函数

```
1 #include <pthread.h>
2
3 int pthread_join(pthread_t thread, void **retval);
4 功能：回收线程的资源（阻塞等子线程结束）
5 参数：
6     @thread:线程号
7     @retval:接收pthread_exit退出时候的线程状态值
8 返回值：成功返回0，失败返回错误码
```

```
1 #include <head.h>
2
3 void* task1(void* arg)
4 {
5     int n = 5;
6     static int num=12345;
7     while (1) {
8         printf("我是子线程\n");
9         sleep(1);
10        n--;
11        if (n == 0)
12            pthread_exit((void *)&num);
13    }
14 }
15 int main(int argc, const char* argv[])
16 {
17     pthread_t tid;
18
19     if ((errno = pthread_create(&tid, NULL, task1, NULL)) != 0)
20         PRINT_ERR("pthread_create error");
21
22     int *retval;
23     pthread_join(tid, (void **)&retval);
24     printf("retval = %d\n", *retval);
25     return 0;
26 }
```

3.3.4多线程拷贝文件的练习

练习：在主线程中创建两个子线程，让这两个子线程同时拷贝文件，各拷贝一半。

./a.out srcfile destfile

```
1 #include <head.h>
2 typedef struct {
3     const char* src;
4     const char* dest;
5     int start;
6     int len;
7 } file_t;
8 int get_file_len(const char* file)
9 {
10     int fd, len;
11     if ((fd = open(file, O_RDONLY)) == -1)
12         PRINT_ERR("open error");
13     len = lseek(fd, 0, SEEK_END);
14     close(fd);
15     return len;
16 }
17 int init_src_file(const char* file)
18 {
19     int fd;
20     if ((fd = open(file, O_WRONLY | O_CREAT | O_TRUNC, 0666)) == -1)
21         PRINT_ERR("open error");
22
23     close(fd);
24     return 0;
25 }
26 int copy_file(const char* src, const char* dest, int start, int len)
27 {
28     char s[20];
29     int fd1, fd2;
30     int ret, count = 0;
31     // 1.以只读方式打开源文件，以只写方式打开目标文件
32     if ((fd1 = open(src, O_RDONLY)) == -1)
33         PRINT_ERR("open src error");
```

```
34     if ((fd2 = open(dest, O_WRONLY)) == -1)
35         PRINT_ERR("open dest error");
36     // 2.定位源和目标文件的光标
37     lseek(fd1, start, SEEK_SET);
38     lseek(fd2, start, SEEK_SET);
39     // 3.循环拷贝
40     while (count < len) {
41         ret = read(fd1, s, sizeof(s)); // 从源文件中读
42         count += ret; // 将每次读的数据加到count中
43         write(fd2, s, ret);
44     }
45     // 4.关闭文件
46     close(fd1);
47     close(fd2);
48     return 0;
49 }
50 void* thread(void* arg)
51 {
52     file_t *f = (file_t *)arg;
53     copy_file(f->src, f->dest, f->start, f->len);
54     pthread_exit(NULL);
55 }
56
57 int main(int argc, const char* argv[])
58 {
59     int len;
60     pid_t pid;
61     // 1.检查参数个数
62     if (argc != 3) {
63         fprintf(stderr, "input error,try again\n");
64         fprintf(stderr, "usage:./a.out srcfile destfile\n");
65         return -1;
66     }
67     // 2.获取源文件大小
68     len = get_file_len(argv[1]);
69
70     // 3.创建出目标文件，并清空
71     init_src_file(argv[2]);
72
73     // 4.创建两个线程拷贝文件
74     pthread_t tid1, tid2;
75     file_t f[] = {
76         [0] = {
77             .src = argv[1],
78             .dest = argv[2],
79             .start = 0,
80             .len = len / 2,
81         },
82         [1] = {
83             .src = argv[1],
84             .dest = argv[2],
85             .start = len/2,
86             .len = (len - len / 2),
87         },
88     };
89     if ((errno = pthread_create(&tid1, NULL, thread, (void *)&f[0]))
90         PRINT_ERR("pthread_create error");
91     if ((errno = pthread_create(&tid2, NULL, thread, (void *)&f[1]))
92         PRINT_ERR("pthread_create error");
93
94     pthread_join(tid1, NULL);
95     pthread_join(tid2, NULL);
96     return 0;
97 }
```

3.3.5pthread_detach函数

线程的状态：

线程本身有两种状态：结合态，分离态。在使用pthread_create函数

创建线程的时候默认就是结合态的线程，结合态的线程需要调用pthread_join

来回收资源，如果将线程标记为分离态，分离态的线程资源会被自动回收，

不需要其他的线程回收其资源。

```
1  int pthread_detach(pthread_t thread);
2  功能：将线程标记为分离态
3  参数：
4      @thread:线程号
5  返回值：成功返回0，失败返回错误码
```

```
1  #include <head.h>
```



```
2
3 void* task1(void* arg)
4 {
5     int n = 5;
6     pthread_detach(pthread_self());
7     while (1) {
8         printf("我是子线程\n");
9         sleep(1);
10        n--;
11        if (n == 0)
12            pthread_exit(NULL);
13    }
14 }
15 int main(int argc, const char* argv[])
16 {
17     pthread_t tid;
18
19     if ((errno = pthread_create(&tid, NULL, task1, NULL)) != 0)
20         PRINT_ERR("pthread_create error");
21
22     sleep(1);
23
24     pthread_join(tid, NULL);
25     printf("0000000000000000\n");
26     return 0;
27 }
```

3.3.6pthread_cancel函数

```
1 int pthread_cancel(pthread_t thread);
2 功能：给thread发送一个取消的信号
3 参数：
4     @thread:线程号
5 返回值：成功返回0，失败返回错误码
6
7 int pthread_setcancelstate(int state, int *oldstate);
8 功能：设置线程是否可被取消
9 参数：
10    @state:
11        PTHREAD_CANCEL_ENABLE:线程可被取消（默认）
12        PTHREAD_CANCEL_DISABLE: 线程不能被取消
13    @oldstate:旧的线程状态
14 返回值：成功返回0，失败返回错误码
15
16 int pthread_setcanceltype(int type, int *oldtype);
17 功能：设置线程取消时机
18 参数：
19    @type:
20        PTHREAD_CANCEL_DEFERRED:延时取消（默认），
21            线程中如果是while(1),就找不到
22            取消点，线程取消将被延时
23        PTHREAD_CANCEL_ASYNCHRONOUS: 立即取消
24    @oldtype:旧的线程类型
25 返回值：成功返回0，失败返回错误码
```

```
1 #include <head.h>
2 pthread_t tid1, tid2;
3 void* task1(void* arg)
4 {
5     sleep(3);
6     pthread_cancel(tid2);
7     while (1);
8 }
9 void* task2(void* arg)
10 {
11     //pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
12     pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
13     while (1) {
14         // printf("我是线程2...\n");
15         // sleep(1);
16     }
17 }
18 int main(int argc, const char* argv[])
19 {
20     if ((errno = pthread_create(&tid1, NULL, task1, NULL)) != 0)
21         PRINT_ERR("pthread_create error");
22     if ((errno = pthread_create(&tid2, NULL, task2, NULL)) != 0)
23         PRINT_ERR("pthread_create error");
24
25     pthread_join(tid1, NULL);
26     pthread_join(tid2, NULL);
27     return 0;
28 }
```

3.4多线程访问全局变量问题

```
1  #include <head.h>
2  // 对于这个代码，如果num变量不加volatile，使用gcc编译
3  // 的使用如果加上-O1 -O2 -O3优化等级（数字越大优化等级越高）
4  // 会发现task2线程不能够退出，因为编译器对num变量进行了优化
5  // 认为num的值一直都是0，所以出现了死循环，为了解决这个问题
6  // 就需要在num变量前加volatile，不让编译器对其优化，每次都能
7  // 得到想要的结果。
8  // 总结：多线程访问全局变量的时候，对变量加volatile。
9  volatile int num = 0;
10 void* task1(void* arg)
11 {
12     sleep(2);
13     num = 1;
14     printf("线程1退出了...\n");
15     pthread_exit(NULL);
16 }
17 void* task2(void* arg)
18 {
19     while (num == 0);
20     printf("线程2退出了...\n");
21     pthread_exit(NULL);
22 }
23 int main(int argc, const char* argv[])
24 {
25     pthread_t tid1, tid2;
26
27     if ((errno = pthread_create(&tid1, NULL, task1, NULL)) != 0)
28         PRINT_ERR("pthread_create error");
29     if ((errno = pthread_create(&tid2, NULL, task2, NULL)) != 0)
30         PRINT_ERR("pthread_create error");
31
32     pthread_join(tid1, NULL);
33     pthread_join(tid2, NULL);
34     return 0;
35 }
```