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

```
pid_t setsid(void);
功能：设置会话id
参数：
    @无
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

```
int chdir(const char *path);
功能：切换目录
参数：
    @path:路径
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

```
#include <sys/types.h>
#include <sys/stat.h>

mode_t umask(mode_t mask);
功能：设置文件的掩码
参数：
    @mask:掩码值
返回值：返回设置前的掩码值，总是会成功
```

5. 创建日志文件

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

6. 文件描述符重定向工作

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

```
#include <unistd.h>

int dup(int oldfd);
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。
    oldfd和newfd都能操作文件，两者共用光标。
参数：
    @oldfd:旧的文件描述符
返回值：成功返回新的文件描述符，失败返回-1置位错误码


int dup2(int oldfd, int newfd);
功能：将oldfd重定向到newfd中，newfd是用户自己指定的，如果newfd
    之前被分配过在使用前会关闭它们。
参数：
    @oldfd:旧的文件描述符
    @newfd:新的文件描述符
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

```
#include <head.h>

int main(int argc, const char* argv[])
{
    int fd;
    if ((fd = open("./daemon.log", O_RDWR | O_CREAT | O_APPEND, 0666)) == -1)
        PRINT_ERR("open error");

    //关闭标准输入，标准输出，标准出错
    close(0);
    close(1);
    close(2);

    //将0,1,2重定向到文件fd中
    dup(fd);
    dup(fd);
    dup(fd);

    printf("hello daemon dup process...\n");
    fflush(stdout);
    fprintf(stderr,"this is test daemon dup stderr...\n");
    lseek(fd,0,SEEK_SET);
    char ch;
    scanf("%c",&ch);
    printf("ch = %c\n",ch);
    return 0;
}
```

1.3.3dup2文件描述符重定向实例

```
#include <head.h>

int main(int argc, const char* argv[])
{
    int fd;
    if ((fd = open("./daemon.log", O_RDWR | O_CREAT | O_APPEND, 0666)) == -1)
        PRINT_ERR("open error");

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

    printf("hello daemon dup process...\n");
    fflush(stdout);
    fprintf(stderr,"this is test daemon dup stderr...\n");
    lseek(fd,0,SEEK_SET);
    char ch;
    scanf("%c",&ch);
    printf("ch = %c\n",ch);
    return 0;
}
```

1.4守护进程创建的实例

```c
#include <head.h>

int main(int argc, const char* argv[])
{
    pid_t pid;
    int fd;
    if ((pid = fork()) == -1) {
        PRINT_ERR("fork error");
    } else if (pid == 0) {
        // 1.孤儿进程
        // 2.设置会话id
        if(setsid()==-1)
            PRINT_ERR("setsid error");
        // 3.切换目录(/var/log/)
        if (chdir("/"))
            PRINT_ERR("chdir error");
        // 4.修改掩码
        umask(0);
        // 5.创建日志文件
        if ((fd = open("./daemon.log", O_RDWR | O_CREAT | O_APPEND, 0666)) == -1)
            PRINT_ERR("open error");
        // 6.文件描述符重定向
        dup2(fd,0);
        dup2(fd,1);
        dup2(fd,2);
        // 7.开启自己的服务
        char s[] = "i am test daemon process\n";
        while(1){
            write(1,s,strlen(s));
            sleep(1);
        }

    } else {
        // 让父进程退出
        exit(0);
    }
    return 0;
}
```

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

```c
#include <stdlib.h>

int system(const char *command);
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
参数：
    @command:可执行程序的路径及名字
返回值：
   *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
   *如果无法创建子进程，或者无法检索其状态，则返回值为-1。
   *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）
   *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

```c
#include <head.h>

int main(int argc, const char* argv[])
{
    // 首先会创建一个子进程，在子进程内执行"/bin/ls"可执行程序
    //  if(system("/bin/ls -l"))
    //      PRINT_ERR("system error");

    // if(system("./b.out"))
    //     PRINT_ERR("system error");

    if (system("./myshell.sh 111 222 333 4444 555"))
        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。

```
ps -ajx ==>看进程附加态（l，代表多线程）
ps -eLf ==>多线程
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

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
功能：创建线程
参数：
    @thread:线程号指针
    @attr:线程属性（一般填写为NULL）
    @start_routine：线程体（指向线程处理函数）
    @arg: 向线程体传递的参数（如果没有填写为NULL）
返回值：成功返回0，失败返回错误码
Compile and link with -pthread.
```

3.2.3线程创建的实例1(不传参)

```
#include <head.h>
void* task1(void* arg)
{
    while (1) {
        printf("我是子线程...\n");
        sleep(1);
    }
}
int main(int argc, const char* argv[])
{
    pthread_t tid;

    if ((errno = pthread_create(&tid, NULL, task1, NULL)) != 0)
        PRINT_ERR("pthread_create error");

    while (1) {
        printf("我是主线程...\n");
        sleep(1);
    }
    return 0;
}
```

3.2.3线程创建的实例2（传参）

```
#include <head.h>
typedef struct {
    char name[20];
    char sex;
    int age;
}stu_t;
void* task1(void* arg)
{
    stu_t *stu = (stu_t *)arg;
```

```
    printf("name=%s,sex=%c,age=%d\n",stu->name,stu->sex,stu->age);
    while (1) {
        printf("我是子线程...\n");
        sleep(1);
    }
}
int main(int argc, const char* argv[])
{
    pthread_t tid;
    stu_t stu = {
        .name = "zhangsan",
        .sex = 'm',
        .age = 20,
    };

    if ((errno = pthread_create(&tid, NULL, task1, (void *)&stu)) != 0)
        PRINT_ERR("pthread_create error");

    while (1) {
        printf("我是主线程...\n");
        sleep(1);
    }
    return 0;
}
```

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

```
#include <head.h>
typedef struct {
    char name[20];
    char sex;
    int age;
} stu_t;

stu_t stu = {
    .name = "zhangsan",
    .sex = 'm',
    .age = 20,
};
void* task1(void* arg)
{
    printf("child:name=%s,sex=%c,age=%d\n", stu.name, stu.sex, stu.age);
    while (1) {
        printf("我是子线程...\n");
        sleep(1);
    }
}
int main(int argc, const char* argv[])
{
    pthread_t tid;
    printf("parent:name=%s,sex=%c,age=%d\n", stu.name, stu.sex, stu.age);
    if ((errno = pthread_create(&tid, NULL, task1,NULL))!=0)
        PRINT_ERR("pthread_create error");

    while (1) {
        printf("我是主线程...\n");
        sleep(1);
    }
    return 0;
}
```

3.2.5线程创建的实例5

```
#include <head.h>

void* task1(void* arg)
{
    printf("我是子线程...\n");
}
int main(int argc, const char* argv[])
{
    pthread_t tid;

    if ((errno = pthread_create(&tid, NULL, task1, NULL)) != 0)
        PRINT_ERR("pthread_create error");

    printf("我是主线程...\n");

    //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
    //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);
    sleep(1);
```

```
    return 0;
}
```

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

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

```
#include <head.h>

void* task1(void* arg)
{
    printf("我是子线程...\n");
    printf("子:tid = %ld\n",pthread_self());
}
int main(int argc, const char* argv[])
{
    pthread_t tid;

    if ((errno = pthread_create(&tid, NULL, task1, NULL)) != 0)
        PRINT_ERR("pthread_create error");

    printf("主:tid = %ld,子:tid = %ld\n",pthread_self(),tid);

    sleep(1);
    return 0;
}
```

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。

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

```
#include <head.h>

void* task1(void* arg)
{
    while(1){
        sleep(1);
    }
}
int main(int argc, const char* argv[])
{
    pthread_t tid;

    if ((errno = pthread_create(&tid, NULL, task1, NULL)) != 0)
        PRINT_ERR("pthread_create error");

    printf("主:tid = %ld,子:tid = %ld\n", pthread_self(), tid);

     int n = 5;
    while (1) {
        printf("我是主线程...\n");
        sleep(1);
        n--;
        if (n == 0)
```

```
            pthread_exit(NULL);
    }
    return 0;
}
```

3.3.3pthread_join函数

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
功能：回收线程的资源（阻塞等子线程结束）
参数：
    @thread:线程号
    @retval:接收pthread_exit退出时候的线程状态值
返回值：成功返回0，失败返回错误码
```

```
#include <head.h>

void* task1(void* arg)
{
    int n = 5;
    static int num=12345;
    while (1) {
        printf("我是子线程\n");
        sleep(1);
        n--;
        if (n == 0)
            pthread_exit((void *)&num);
    }
}
int main(int argc, const char* argv[])
{
    pthread_t tid;

    if ((errno = pthread_create(&tid, NULL, task1, NULL)) != 0)
        PRINT_ERR("pthread_create error");

    int *retval;
    pthread_join(tid,(void **)&retval);
    printf("retval = %d\n",*retval);
    return 0;
}
```

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

```
#include <head.h>
typedef struct {
    const char* src;
    const char* dest;
    int start;
    int len;
} file_t;
int get_file_len(const char* file)
{
    int fd, len;
    if ((fd = open(file, O_RDONLY)) == -1)
        PRINT_ERR("open error");
    len = lseek(fd, 0, SEEK_END);
    close(fd);
    return len;
}
int init_src_file(const char* file)
{
    int fd;
    if ((fd = open(file, O_WRONLY | O_CREAT | O_TRUNC, 0666)) == -1)
        PRINT_ERR("open error");

    close(fd);
    return 0;
}
int copy_file(const char* src, const char* dest, int start, int len)
{
    char s[20];
    int fd1, fd2;
    int ret, count = 0;
    // 1.以只读方式打开源文件，以只写方式打开目标文件
    if ((fd1 = open(src, O_RDONLY)) == -1)
        PRINT_ERR("open src error");
```

```
    if ((fd2 = open(dest, O_WRONLY)) == -1)
        PRINT_ERR("open dest error");
    // 2.定位源和目标文件的光标
    lseek(fd1, start, SEEK_SET);
    lseek(fd2, start, SEEK_SET);
    // 3.循环拷贝
    while (count < len) {
        ret = read(fd1, s, sizeof(s)); // 从源文件中读
        count += ret; // 将每次读的数据加到count中
        write(fd2, s, ret);
    }
    // 4.关闭文件
    close(fd1);
    close(fd2);
    return 0;
}
void* thread(void* arg)
{
    file_t *f = (file_t *)arg;
    copy_file(f->src, f->dest, f->start, f->len);
    pthread_exit(NULL);
}

int main(int argc, const char* argv[])
{
    int len;
    pid_t pid;
    // 1.检查参数个数
    if (argc != 3) {
        fprintf(stderr, "input error,try again\n");
        fprintf(stderr, "usage:./a.out srcfile destfile\n");
        return -1;
    }
    // 2.获取源文件大小
    len = get_file_len(argv[1]);

    // 3.创建出目标文件，并清空
    init_src_file(argv[2]);

    // 4.创建两个线程拷贝文件
    pthread_t tid1, tid2;
    file_t f[] = {
        [0] = {
            .src = argv[1],
            .dest = argv[2],
            .start = 0,
            .len = len / 2,
        },
        [1] = {
            .src = argv[1],
            .dest = argv[2],
            .start = len/2,
            .len = (len - len / 2),
        },
    };
    if ((errno = pthread_create(&tid1, NULL, thread, (void *)&f[0])))
        PRINT_ERR("pthread_create error");
    if ((errno = pthread_create(&tid2, NULL, thread, (void *)&f[1])))
        PRINT_ERR("pthread_create error");

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

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。

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

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

```c

void* task1(void* arg)
{
    int n = 5;
    pthread_detach(pthread_self());
    while (1) {
        printf("我是子线程\n");
        sleep(1);
        n--;
        if (n == 0)
            pthread_exit(NULL);
    }
}
int main(int argc, const char* argv[])
{
    pthread_t tid;

    if ((errno = pthread_create(&tid, NULL, task1, NULL)) != 0)
        PRINT_ERR("pthread_create error");

    sleep(1);

    pthread_join(tid,NULL);
    printf("OOOOOOOOOOOOOOOOOO\n");
    return 0;
}
```

3.3.6pthread_cancel函数

```
int pthread_cancel(pthread_t thread);
功能：给thread发送一个取消的信号
参数：
    @thread:线程号
返回值：成功返回0，失败返回错误码

int pthread_setcancelstate(int state, int *oldstate);
功能：设置线程是否可被取消
参数：
    @state:
        PTHREAD_CANCEL_ENABLE:线程可被取消（默认）
        PTHREAD_CANCEL_DISABLE：线程不能被取消
    @oldstate:旧的线程状态
返回值：成功返回0，失败返回错误码

int pthread_setcanceltype(int type, int *oldtype);
功能：设置线程取消时机
参数：
    @type:
        PTHREAD_CANCEL_DEFERRED:延时取消（默认），
                    线程中如果是while(1),就找不到
                    取消点，线程取消将被延时
        PTHREAD_CANCEL_ASYNCHRONOUS：立即取消
    @oldtype:旧的线程类型
返回值：成功返回0，失败返回错误码
```

```c
#include <head.h>
pthread_t tid1, tid2;
void* task1(void* arg)
{
    sleep(3);
    pthread_cancel(tid2);
    while (1);
}
void* task2(void* arg)
{
    //pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL);
    while (1) {
        // printf("我是线程2...\n");
        // sleep(1);
    }
}
int main(int argc, const char* argv[])
{
    if ((errno = pthread_create(&tid1, NULL, task1, NULL)) != 0)
        PRINT_ERR("pthread_create error");
    if ((errno = pthread_create(&tid2, NULL, task2, NULL)) != 0)
        PRINT_ERR("pthread_create error");

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

3.4多线程访问全局变量问题

```c
#include <head.h>
// 对于这个代码，如果num变量不加volatile，使用gcc编译
// 的使用如果加上-O1 -O2 -O3优化等级（数字越大优化等级越高）
// 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
// 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
// 就需要在num变量前加volatile，不让编译器对其优化，每次都能
// 得到想要的结果。
// 总结：多线程访问全局变量的时候，对变量加volatile。
volatile int num = 0;
void* task1(void* arg)
{
    sleep(2);
    num = 1;
    printf("线程1退出了...\n");
    pthread_exit(NULL);
}
void* task2(void* arg)
{
    while (num == 0);
    printf("线程2退出了...\n");
    pthread_exit(NULL);
}
int main(int argc, const char* argv[])
{
    pthread_t tid1, tid2;

    if ((errno = pthread_create(&tid1, NULL, task1, NULL)) != 0)
        PRINT_ERR("pthread_create error");
    if ((errno = pthread_create(&tid2, NULL, task2, NULL)) != 0)
        PRINT_ERR("pthread_create error");

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```