1.线程间通信

线程间通信一般在指线程互斥机制和线程的同步机制。

1.1多线程产生竞态的问题

当多个线程同时使用同一个全局变量的时候，因为两个线程都想

操作这个变量，此时竞态就会产生了，解决竞态的方式就可以使用

线程的互斥锁。

```c
#include <head.h>
volatile int money = 10000;
// 张三
void* task1(void* arg)
{
    while (1) {
        money -= 50;
        if (money >= 0) {
            printf("张三成功取了50块钱，余额=%d\n", money);
        } else {
            money += 50;
            printf("张三取钱失败，余额不足\n");
            pthread_exit(NULL);
        }
        sleep(1);
    }
}
// 李四
void* task2(void* arg)
{
    while (1) {
        money -= 100;
        if (money >= 0) {
            printf("李四成功取了100块钱，余额=%d\n", money);
```

```
        } else {
            money += 100;
            printf("李四取钱失败，余额不足\n");
            pthread_exit(NULL);
        }
        sleep(1);
    }
}
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;
}
```

```
linux@ubuntu:~/work/day7$ ./a.out
李四成功取了100块钱，余额=9900
张三成功取了50块钱，余额=9850
李四成功取了100块钱，余额=9750
张三成功取了50块钱，余额=9700
李四成功取了100块钱，余额=9600
张三成功取了50块钱，余额=9550
李四成功取了100块钱，余额=9450
张三成功取了50块钱，余额=9400
张三成功取了50块钱，余额=9350
李四成功取了100块钱，余额=9250
李四成功取了100块钱，余额=9150
张三成功取了50块钱，余额=9200
张三成功取了50块钱，余额=9150
李四成功取了100块钱，余额=9150
李四成功取了100块钱，余额=9050
```

1.2线程的互斥锁

1.2.1线程互斥锁的API

```
1.定义互斥锁
    pthread_mutex_t mutex;
2.初始化线程互斥锁
    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
    //静态初始化
    int pthread_mutex_init(pthread_mutex_t * mutex,
                           const pthread_mutexattr_t * attr);
    //动态初始化
    功能：初始化互斥锁
    参数：
        @mutex:被初始化的锁
        @attr:锁的属性，一般填写为NULL(默认属性)
    返回值：成功返回0，失败返回错误码
3.上锁
     int pthread_mutex_trylock(pthread_mutex_t *mutex);
     //尝试获取锁，如果锁资源存在那就占用锁，如果锁资源不可利用，立即返回。
     int pthread_mutex_lock(pthread_mutex_t *mutex);
     功能：上锁（如果线程获取不到锁的资源，线程阻塞，直到其他的线程将锁释放）
     参数：
          @mutex:执行锁的指针
     返回值：成功返回0，失败返回错误码
4.解锁
```

```
    int pthread_mutex_unlock(pthread_mutex_t *mutex);
    功能：解锁
    参数：
          @mutex:执行锁的指针
     返回值：成功返回0，失败返回错误码
5.销毁锁
    int pthread_mutex_destroy(pthread_mutex_t *mutex);
    功能：销毁互斥锁
    参数：
        @mutex:执行锁的指针
    返回值：成功返回0，失败返回错误码
```

1.2.2线程互斥锁的实例

```c
#include <head.h>
pthread_mutex_t lock; // 定义锁

volatile int money = 10000;  //临界资源
// 张三
void* task1(void* arg)
{
    while (1) {
        pthread_mutex_lock(&lock);
        money -= 50;
        if (money >= 0) {
            printf("张三成功取了50块钱,余额=%d\n", money);
        } else {
            money += 50;
            printf("张三取钱失败,余额不足\n");
            pthread_mutex_unlock(&lock);
            pthread_exit(NULL);
        }
        // sleep(1);
        pthread_mutex_unlock(&lock);
    }
}
// 李四
void* task2(void* arg)
{
    while (1) {
        pthread_mutex_lock(&lock);
        money -= 100;
        if (money >= 0) {
            printf("李四成功取了100块钱,余额=%d\n", money);
        } else {
            money += 100;
            printf("李四取钱失败,余额不足\n");
            pthread_mutex_unlock(&lock);
            pthread_exit(NULL);
        }
        // sleep(1);
        pthread_mutex_unlock(&lock);
    }
}
int main(int argc, const char* argv[])
{
    pthread_t tid1, tid2;

    // 初始化互斥锁
    pthread_mutex_init(&lock, NULL);

    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);

    // 销毁锁
    pthread_mutex_destroy(&lock);
    return 0;
}
```

1.2.3线程互斥锁死锁问题

1. 产生死锁的四个必要条件

   互斥，请求保持，不可剥夺，循环等待

2. 死锁的规避方法

   1.指定线程获取锁的顺序

2.尽量避免锁的嵌套使用

3.给线程上锁指定超时时间（pthread_mutex_timedlock）

4.在全局位置指定锁是否被使用的状态，如果被使用就不在获取

1.3无名信号量

1.3.1无名信号的工作原理

当多个线程在访问全局变量的时候如果使用互斥锁只能保证有一个线程

在操作临界资源，不能保证线程的执行的先后顺序，如果想要控制线程的

执行顺序就可以使用无名信号量完成，无名信号量是实现线程同步的机制。

线程同步机制一般使用在生产者和消费者模型上。

1.3.2无名信号量的API

```
#include <semaphore.h>
1.定义无名信号量
    sem_t sem;
2.初始化无名信号量
    int sem_init(sem_t *sem, int pshared, unsigned int value);
    功能：初始化无名信号量
    参数：
        @sem:指向无名信号量的指针
        @pshared:0 线程的同步
                 1 进程的同步（亲缘关系进程）
        @value:信号的初值  1
    返回值：成功返回0，失败返回-1置位错误码
3.获取信号量（P操作）
    int sem_wait(sem_t *sem);
    功能：申请资源（让信号量的值减去1，然后和0比较如果结果为0，表示获取锁成功了）
        如果在调用sem_wait的时候获取不到资源，sem_wait会阻塞
    参数：
        @sem:指向无名信号量的指针
    返回值：成功返回0，失败返回-1置位错误码
4.释放信号量（V操作）
    int sem_post(sem_t *sem);
    功能：释放资源
    参数：
        @sem:指向无名信号量的指针
    返回值：成功返回0，失败返回-1置位错误码
5.销毁无名信号量
    int sem_destroy(sem_t *sem);
    功能：销毁无名信号量
    参数：
        @sem:指向无名信号量的指针
    返回值：成功返回0，失败返回-1置位错误码
```

1.3.3无名信号量实例

```
#include <head.h>
sem_t sem1, sem2; // 定义无名信号量
// 生产者线程
void* task1(void* arg)
{
    while (1) {
        sem_wait(&sem1); //申请资源sem1
        printf("我生成了一部手机...\n");
        sem_post(&sem2); // 释放资源 sem2
    }
}
// 消费者线程
void* task2(void* arg)
{
    while (1) {
        sem_wait(&sem2); // 申请资源 sem2
        printf("我购买了一部手机\n");
        sem_post(&sem1); //释放资源 sem1
    }
}
int main(int argc, const char* argv[])
{
    pthread_t tid1, tid2;

    // 初始化无名信号量
    sem_init(&sem1, 0, 1);
    sem_init(&sem2, 0, 0);

    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);

    // 销毁无名信号量
    sem_destroy(&sem1);
    sem_destroy(&sem2);
    return 0;
}
```

1.3.4无名信号量的练习

练习：有三个线程分别打印ABC，使用无名信号量实

现三个线程同步，逐次打印ABCABCABC...

```
#include <head.h>
sem_t sem1, sem2, sem3; // 定义无名信号量
// A
void* task1(void* arg)
{
    while (1) {
        sem_wait(&sem1); // 申请资源sem1
        printf("A");
        sem_post(&sem2); // 释放资源 sem2
    }
}
// B
void* task2(void* arg)
{
    while (1) {
        sem_wait(&sem2); // 申请资源 sem2
        printf("B");
        sem_post(&sem3); // 释放资源 sem3
    }
}
// C
void* task3(void* arg)
{
    while (1) {
        sem_wait(&sem3); // 申请资源 sem2
        printf("C\n");
        sleep(1);
        sem_post(&sem1); // 释放资源 sem1
    }
}
int main(int argc, const char* argv[])
{
    pthread_t tid1, tid2,tid3;

    // 初始化无名信号量
    sem_init(&sem1, 0, 1);
    sem_init(&sem2, 0, 0);
    sem_init(&sem3, 0, 0);

    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");
    if ((errno = pthread_create(&tid3, NULL, task3, NULL)) != 0)
        PRINT_ERR("pthread_create error");

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);

    // 销毁无名信号量
    sem_destroy(&sem1);
    sem_destroy(&sem2);
    sem_destroy(&sem3);
    return 0;
}
```

1.4条件变量

1.4.1条件变量的工作原理

条件变量也是线程的同步机制，条件变量更适合用在多个线程的同步工作。

1.4.2条件变量的API

```
1.定义条件变量
    pthread_cond_t cond;

2.初始化条件变量
    pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
    //静态初始化
    int pthread_cond_init(pthread_cond_t * cond,const pthread_condattr_t * attr);
    功能：动态初始化一个条件变量
    参数：
       @cond：条件变量的指针
       @attr:NULL使用默认属性
    返回值：成功返回0，失败返回非0

3.阻塞等待条件变量
    int pthread_cond_wait(pthread_cond_t * cond,pthread_mutex_t * mutex);
    功能：阻塞等待条件变量，在条件变量中维护了一个队列，这里的互斥锁就是为
         了解决在往队列中放线程的时候出现竞态问题的。
    使用的步骤：
        1.使用pthread_mutex_lock上锁
        2.调用pthread_cond_wait
            2.1将当前线程放入队列
            2.2解锁
            2.3休眠
            2.4获取锁
            2.5休眠状态退出
        3.你的程序
        4.使用pthread_mutex_unlock解锁
    参数：
        @cond:条件变量的地址
        @mutex:互斥锁
    返回值：成功返回0，失败返回非零


4.给休眠的线程发信号或者广播
    int pthread_cond_signal(pthread_cond_t *cond);
    功能：唤醒(至少)一个休眠的线程
    参数：
        @cond:条件变量的地址
    返回值：成功返回0，失败返回非零
    int pthread_cond_broadcast(pthread_cond_t *cond);
    功能：唤醒所有休眠的线程
    参数：
        @cond:条件变量的地址
    返回值：成功返回0，失败返回非零

5.销毁条件变量
    int pthread_cond_destroy(pthread_cond_t *cond);
    功能：销毁条件变量
    参数：
         @cond:条件变量的地址
    返回值：成功返回0，失败返回非零
```

1.4.3条件变量的实例

一个生成这和多个消费者线程同步：

```c
#include <head.h>
pthread_cond_t cond; //定义条件变量
pthread_mutex_t lock; //定义互斥锁
// 生成者
void* task1(void* arg)
{
    while (1) {
        sleep(1);
        printf("我生产了一部手机\n");
        // pthread_cond_signal(&cond); //释放资源
        pthread_cond_broadcast(&cond);
    }
}
// 消费者
void* task2(void* arg)
{
    while (1) {
        pthread_mutex_lock(&lock);
        pthread_cond_wait(&cond,&lock);
```

```
        printf("%#lx购买了一部手机\n",pthread_self());
        pthread_mutex_unlock(&lock);
    }
}
int main(int argc, const char* argv[])
{
    pthread_t tid1, tid2, tid3, tid4, tid5;
    pthread_mutex_init(&lock,NULL);//初始化互斥锁
    pthread_cond_init(&cond,NULL); //初始化条件变量

    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");
    if ((errno = pthread_create(&tid3, NULL, task2, NULL)) != 0)
        PRINT_ERR("pthread_create error");
    if ((errno = pthread_create(&tid4, NULL, task2, NULL)) != 0)
        PRINT_ERR("pthread_create error");
    if ((errno = pthread_create(&tid5, NULL, task2, NULL)) != 0)
        PRINT_ERR("pthread_create error");

    printf("tid2=%#lx,tid3=%#lx,tid3=%#lx,tid5=%#lx\n",
        tid2, tid3, tid4, tid5);

    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    pthread_join(tid3, NULL);
    pthread_join(tid4, NULL);
    pthread_join(tid5, NULL);

    pthread_cond_destroy(&cond);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

（一个）生产者线程和消费者线程同步

```
#include <head.h>
pthread_cond_t cond; // 定义条件变量
pthread_mutex_t lock; // 定义互斥锁
int flags = 0; // 标志位变量

//情况1:  B(pthread_cond_wait) A    A(pthread_cond_wait) B
//情况2:  B(pthread_cond_wait) A B
//情况3:  A A(pthread_cond_wait) B
//情况4:  A B

// 生成者 A
void* task1(void* arg)
{
    while (1) {
        pthread_mutex_lock(&lock);
        if (flags == 1)
            pthread_cond_wait(&cond, &lock);
        printf("我生产了一部手机\n");
        pthread_cond_signal(&cond); // 释放资源
        flags = 1;
        pthread_mutex_unlock(&lock);
    }
}
// 消费者 B
void* task2(void* arg)
{
    while (1) {
        pthread_mutex_lock(&lock);
        if (flags == 0)
            pthread_cond_wait(&cond, &lock);
        printf("购买了一部手机\n");
        pthread_cond_signal(&cond); // 释放资源
        flags = 0;
        pthread_mutex_unlock(&lock);
    }
}
int main(int argc, const char* argv[])
{
    pthread_t tid1, tid2;
    pthread_mutex_init(&lock, NULL); // 初始化互斥锁
    pthread_cond_init(&cond, NULL); // 初始化条件变量

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

    if ((errno = pthread_create(&tid2, NULL, task2, NULL)) != 0)
```

```
47          PRINT_ERR("pthread_create error");
48
49      pthread_join(tid1, NULL);
50      pthread_join(tid2, NULL);
51
52      pthread_cond_destroy(&cond);
53      pthread_mutex_destroy(&lock);
54      return 0;
55  }
```

1.4.4条件变量使用场景

无名信号量适合在线程数比较少的线程中实现同步过程，而条件变量适合在大量线程实现同步过程。例如条件变量的使用场景如下：比如你要编写一个12306买票的服务器当客户端访问服务器的时候，服务器会创建一个线程服务于这个用户。如果有多个用户同时想买票，此时服务需要在瞬间创建一堆线程，这个时间比较长，对用户的体验感不好。所以12306服务器是在启动的时候都已经创建好一堆线程。调用pthread_cond_wait让这些线程休眠，当有客户端请求买票的时候，只需要唤醒这些休眠的线程即可，由于省去了创建线程的时间，所以这种方式的效率非常的高。

2.进程间的通信方式

2.1进程间通信方式简介

在linux系统上常用的进程间通信方式有如下7种：

（1）传统进程间通信

1.无名管道

2.有名管道

3.信号

（2）system V IPC进程间通信

1.消息队列

2.共享内存

3.信号量（信号灯集）

（3）BSD(伯克利分校)基于网络的进程间通信

1.socket实现进程间通信

2.2无名管道

2.2.1无名管道通信原理

无名管道通信原理：如果A和B进程想要通过无名管道通信，那就必须在内核空间创建一个无名管道（64K）,A和B进程必须是亲缘关系的进程，A进程向管道的一端写数据，B进程可以从管道的另外一端读数据。在A进程和B进程进行数据传输的时候是不允许使用lseek函数的。无名管道是半双工的通信方式。如果A进程一直向管道中写数据写满64K的时候A进程阻塞，直到B进程读一部分数据之后A才能继续写。如果B进程在读数据的时候，无名管道是空的，B进程阻塞。

单工：

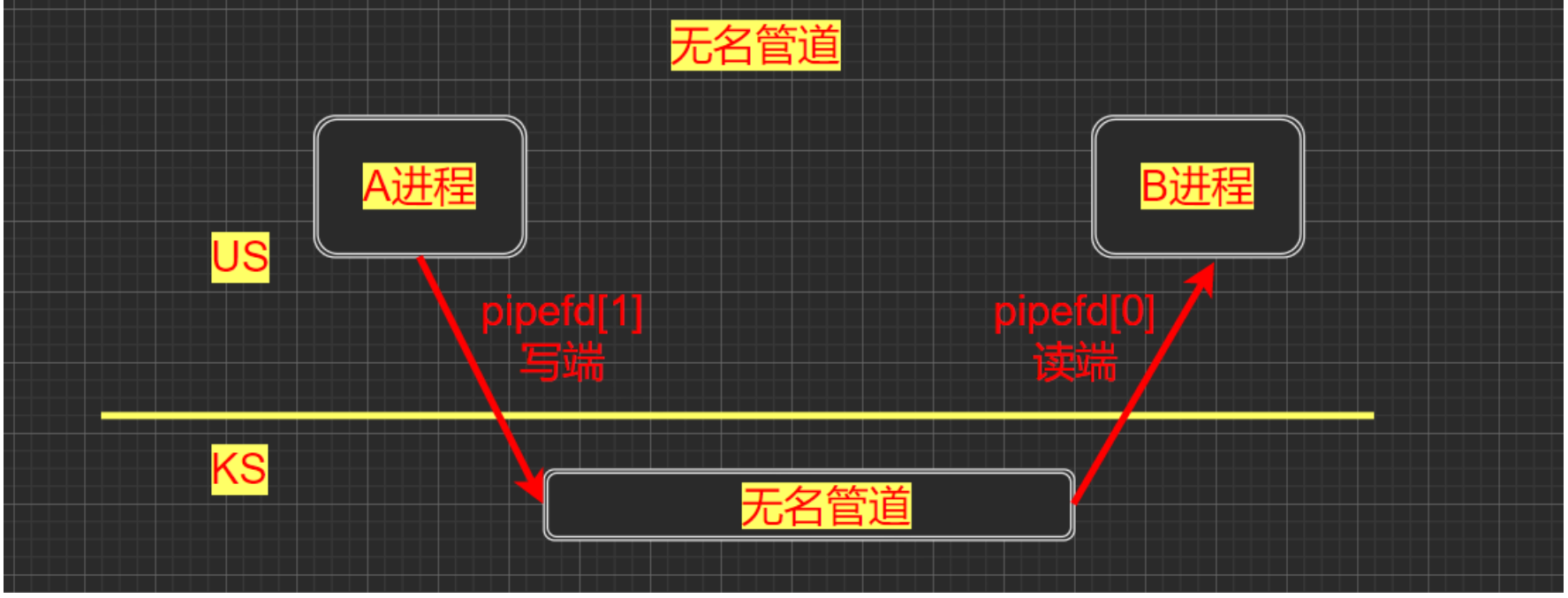
A-------->B

半双工：在同一时刻只能一边发另一边收

A------->B B----->A

全双工：在同一时刻两边可以同时收发

A<----->B



2.2.2无名管道的API

```
#include <unistd.h>

int pipe(int pipefd[2]);
功能：创建一个无名管道
参数：
    @pipefd:返回管道的两端
            pipefd[1]:写端
            pipefd[0]:读端
返回值：成功返回0，失败返回-1置位错误码
```

2.2.3无名管道通信实例

```c
#include <head.h>

int main(int argc, const char* argv[])
{
    pid_t pid;
    int pipefd[2];
    char s[128] = { 0 };
    // 1.创建无名管道
    if (pipe(pipefd))
        PRINT_ERR("pipe error");

    // 2.创建父子进程
    if ((pid = fork()) == -1) {
        PRINT_ERR("fork error");
    } else if (pid == 0) {
        close(pipefd[1]); // 关闭子进程的写端
        // 子进程
        while (1) {
            // 清空s
            memset(s,0,sizeof(s));
            // 从管道向s中读数据
            read(pipefd[0], s, sizeof(s));
            // 如果读到的是quit就退出
            if (strcmp(s, "quit") == 0)
                break;
            // 将读取到的数据打印到终端上
            printf("%s\n", s);
        }
        close(pipefd[0]);
        exit(EXIT_SUCCESS);
    } else {
        close(pipefd[0]); // 关闭父进程的读端
        // 父进程
        while (1) {
            // 从终端向s数组读取字符串
            fgets(s, sizeof(s), stdin);
            // 清除换行符
            if (s[strlen(s) - 1] == '\n')
                s[strlen(s) - 1] = '\0';
            // 向管道中写数据
            write(pipefd[1], s, strlen(s));
            // 如果输入的是quit让进程退出
            if (strcmp(s, "quit") == 0)
                break;
        }
        close(pipefd[1]);
        //等待回收子进程的资源
        wait(NULL);
    }
    return 0;
```

```
}
```

2.2.4无名管道通信特点

读端存在写管道：有多少写多少，直到写满（64K）为止，写阻塞

```c
#include <head.h>

int main(int argc, const char* argv[])
{
    int pipefd[2];
    char s[128] = { 0 };

    // 1.创建无名管道
    if (pipe(pipefd))
        PRINT_ERR("pipe error");
    // 2.读端存在，但是没有读数据，写管道
    // 管道的大小是64K(65536),如果写65536个字节，写没有阻塞，printf("1111111...")会打印
    // 如果将循环的次数改为65537的时候，最后一次写，写阻塞，所以printf("11111...")不会打印
    // 说明写满了
    char ch='a';
    for(int i=0;i<65537;i++){
        write(pipefd[1],&ch,1);
    }

    printf("1111111111111111111111111111\n");
    return 0;
}
```

读端不存在写管道：管道破裂，进程收到SIGPIPE信号，进程退出

```c
#include <head.h>

int main(int argc, const char* argv[])
{
    int pipefd[2];
    char s[128] = { 0 };

    // 1.创建无名管道
    if (pipe(pipefd))
        PRINT_ERR("pipe error");

    close(pipefd[0]); //关闭读端

    char ch='a';
    //关闭读端，写管道，管道破裂，操作系统给当前进程发送SIGPIPE，
    // 将当前进程杀死
    write(pipefd[1],&ch,1);

    while(1);

    return 0;
}
```

写端存在读管道：管道中有多少字节的数据就能读多少数据，如果没有数据的时候，读阻塞

写端不存在端管道：管道中有多少字节的数据就能读多少数据，如果没有数据的时候，读立即返回

```c
#include <head.h>

int main(int argc, const char* argv[])
{
    int pipefd[2];
    char s[128] = "i am test pipe func...\n";

    // 1.创建无名管道
    if (pipe(pipefd))
        PRINT_ERR("pipe error");

    write(pipefd[1],s,strlen(s));

    close(pipefd[1]); //关闭写端

    char ch;
    while(1){
        read(pipefd[0],&ch,1);
        printf("ch = %c\n",ch);
        usleep(100000);
    }
    return 0;
}
```

2.3有名管道

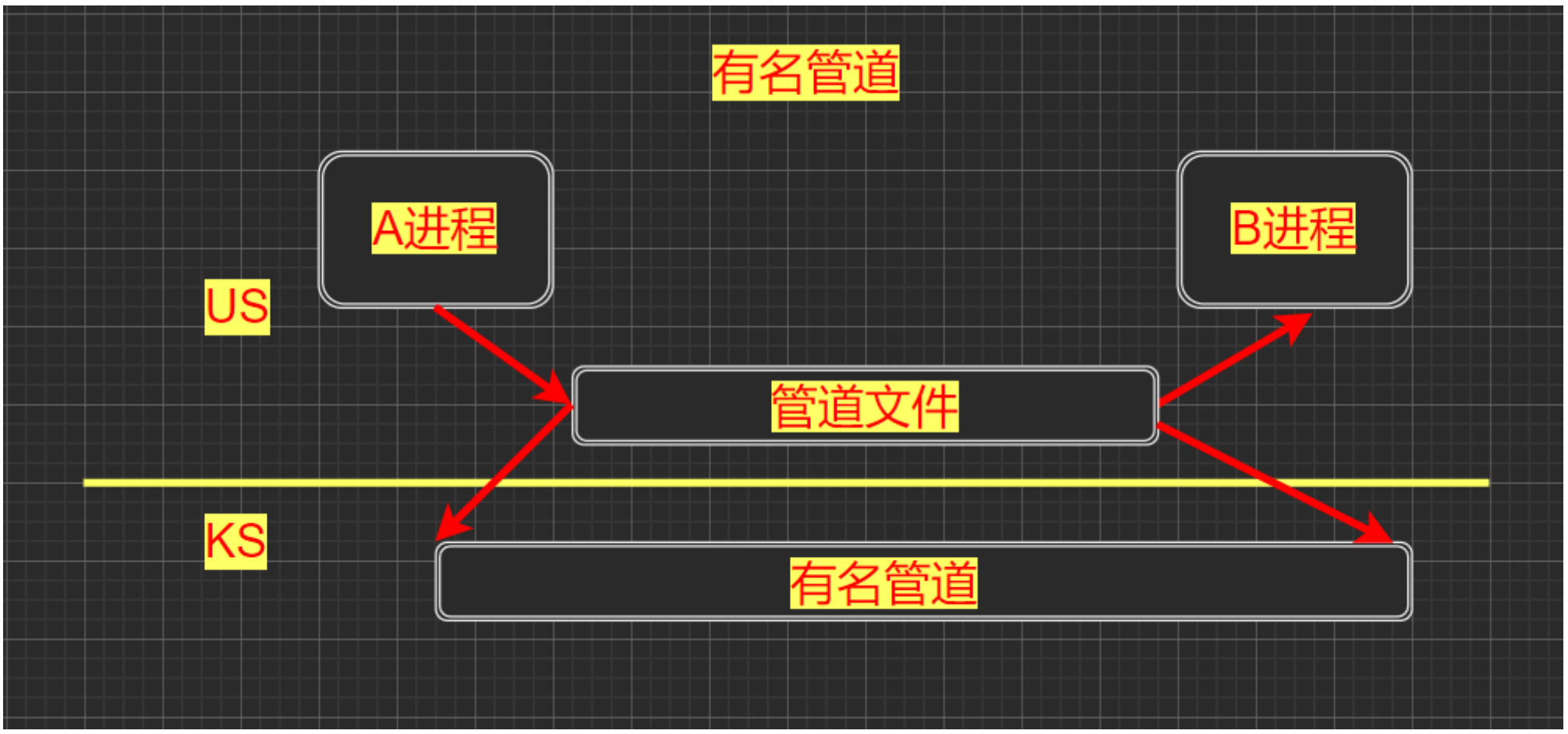
2.3.1有名管道通信原理

有名管道可以实现任意进程间的通信，有名管道的大小也是64K，有名管道也是不支持lseek，

有名管道也是半双工的通信方式。有名管道创建之后会在**用户空间产生一个管道文件**，这个

管道文件是在内存上存储的。如果A和B两个进程想要通过有名管道通信，就打开管道文件，

向管道中写向管道中读就可



2.3.2有名管道的API

```
int mkfifo(const char *pathname, mode_t mode);
功能：创建有名管道
参数：
    @pathname:管道文件的路径及名字
    @mode:管道文件的操作权限(mode & ~umask)
返回值：成功返回0，失败返回-1置位错误码
```

2.3.3有名管道通信实例

01mkfifo.c

```c
#include <head.h>
#define FIFO_NAME "./myfifo"
int main(int argc,const char * argv[])
{
    // 1.创建管道
    if(mkfifo(FIFO_NAME,0666))
        PRINT_ERR("mkfifo error");

    // 2.等待用户使用
    getchar();

    // 3.销毁管道
    char s[50] = {0};
    snprintf(s,sizeof(s),"rm -rf %s",FIFO_NAME);
    system(s);

    return 0;
}
```

02write.c

```c
#include <head.h>
#define FIFO_NAME "./myfifo"
int main(int argc, const char* argv[])
{
    int fd;
    char s[128] = { 0 };
    if ((fd = open(FIFO_NAME, O_WRONLY)) == -1)
        PRINT_ERR("open error");

    while (1) {
        printf("input > ");
        // 从终端向s数组读取字符串
```

```
        fgets(s, sizeof(s), stdin);
        // 清除换行符
        if (s[strlen(s) - 1] == '\n')
            s[strlen(s) - 1] = '\0';
        // 向管道中写数据
        write(fd, s, strlen(s));
        // 如果输入的是quit让进程退出
        if (strcmp(s, "quit") == 0)
            break;
    }

    close(fd);

    return 0;
}
```

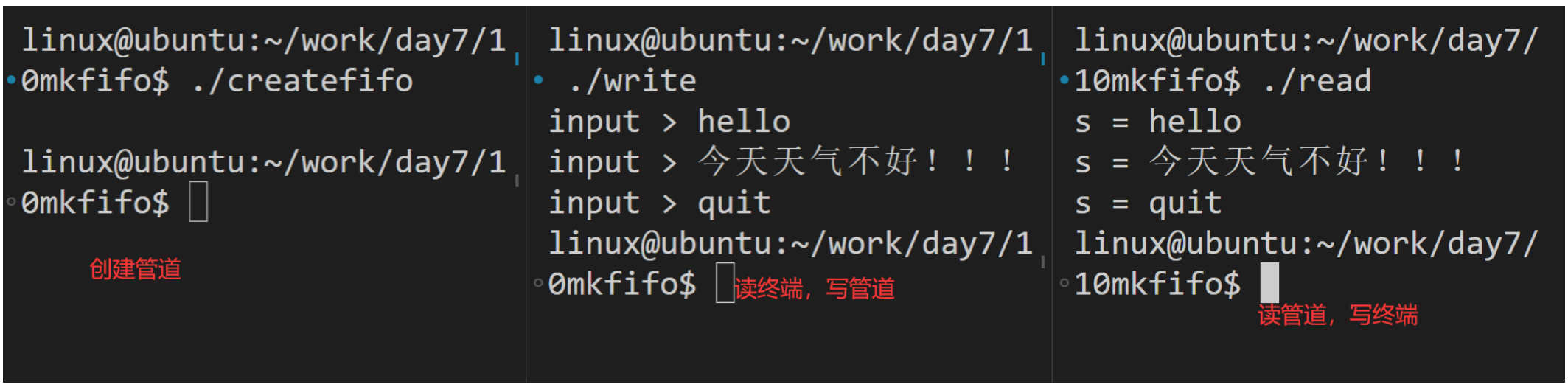
03read.c

```
#include <head.h>
#define FIFO_NAME "./myfifo"
int main(int argc, const char* argv[])
{
    int fd;
    char s[128] = { 0 };
    if ((fd = open(FIFO_NAME, O_RDONLY)) == -1)
        PRINT_ERR("open error");

    while (1) {
        memset(s, 0, sizeof(s));
        read(fd, s, sizeof(s));
        printf("s = %s\n", s);
        if (strcmp(s, "quit") == 0)
            break;
    }

    close(fd);

    return 0;
}
```



2.3.4有名管道的练习

练习：使用有名管道传输文件，要求如下：

A进程读文件，将文件中的内容写入到管道中

B进程读取管道，将管道中的数据写入到新文件中。

01mkfifo.c

```
#include <head.h>
#define FIFO_NAME "./myfifo"
int main(int argc,const char * argv[])
{
    // 1.创建管道
    if(mkfifo(FIFO_NAME,0666))
        PRINT_ERR("mkfifo error");

    // 2.等待用户使用
    getchar();

    // 3.销毁管道
    char s[50] = {0};
    snprintf(s,sizeof(s),"rm -rf %s",FIFO_NAME);
    system(s);

    return 0;
}
```

02write.c

```c
#include <head.h>
#define FIFO_NAME "./myfifo"
int main(int argc, const char* argv[])
{
    int fd1, fd2, ret;
    char s[128] = { 0 };
    // 校验参数
    if (argc != 2) {
        fprintf(stderr, "input error,try again\n");
        fprintf(stderr, "usgage: ./a.out srcfile\n");
        return -1;
    }
    // 打开源文件
    if ((fd1 = open(argv[1], O_RDONLY)) == -1)
        PRINT_ERR("open error");
    // 打开管道文件
    if ((fd2 = open(FIFO_NAME, O_WRONLY)) == -1)
        PRINT_ERR("open error");

    // 循环读写（从源文件读，向管道写）
    while ((ret = read(fd1, s, sizeof(s))) > 0) {
        write(fd2,s,ret);
    }

    close(fd1);
    close(fd2);

    return 0;
}
```

03read.c

```c
#include <head.h>
#define FIFO_NAME "./myfifo"
int main(int argc, const char* argv[])
{
    int fd1, fd2, ret;
    char s[128] = { 0 };
    // 校验参数
    if (argc != 2) {
        fprintf(stderr, "input error,try again\n");
        fprintf(stderr, "usgage: ./a.out destfile\n");
        return -1;
    }
    // 打开目标文件
    if ((fd1 = open(argv[1], O_WRONLY|O_CREAT|O_TRUNC,0666)) == -1)
        PRINT_ERR("open error");
    // 打开管道文件
    if ((fd2 = open(FIFO_NAME, O_RDONLY)) == -1)
        PRINT_ERR("open error");

    // 循环读写（从源文件读，向管道写）
    while ((ret = read(fd2, s, sizeof(s))) > 0) {
        write(fd1,s,ret);
    }

    close(fd1);
    close(fd2);

    return 0;
}
```

2.3.5有名管道读写的特点

读端存在写管道：*有多少写多少，直到写满为止（64K），写阻塞*

读端没有打开过，写管道：*写端在open位置阻塞*

读端先打开后关闭，写管道：*管道破裂，收到SIGPIPE信号，进程结束*

写端存在读管道：*有多少读多少，没数据读的时候读阻塞*

写端没有打开过，读管道：*读端在open的位置阻塞*

写端先打开后关闭，读管道：*有多少读多少，没数据的时候读立即返回（读返回值是0）*

2.4信号

2.4.1信号简介

用户可以通过kill命令给进程发信号，操作系统也可以给进程发送信号。

进程对信号的响应方式有三种：**忽略，默认，捕捉**

2.4.2发信号的命令

kill -信号号 PID

2.4.3信号的查看方式

```
linux@ubuntu:~/work$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

2.4.4常用的信号

信号名	含义	默认操作
SIGHUP	该信号在用户终端关闭时产生，通常是发给和该终端关联的会话内的所有进程	终止
SIGINT	该信号在用户键入INTR字符(Ctrl-C)时产生，内核发送此信号送到当前终端的所有前台进程	终止
SIGQUIT	该信号和SIGINT类似，但由QUIT字符(通常是Ctrl-\)来产生	终止
SIGILL	该信号在一个进程企图执行一条非法指令时产生	终止
SIGSEV	该信号在非法访问内存时产生，如野指针、缓冲区溢出	终止
SIGPIPE	当进程往一个没有读端的管道中写入时产生，代表“管道断裂”	终止

信号名	含义	默认操作
SIGKILL	该信号用来结束进程，并且不能被捕捉和忽略	终止
SIGSTOP	该信号用于暂停进程，并且不能被捕捉和忽略	暂停进程
SIGTSTP	该信号用于暂停进程，用户可键入SUSP字符(通常是Ctrl-Z)发出这个信号	暂停进程
SIGCONT	该信号让进程进入运行态	继续运行
SIGALRM	该信号用于通知进程定时器时间已到	终止
SIGUSR1/2	该信号保留给用户程序使用	终止

SIGCHLD:当子进程退出的时候，父进程会收到这个信号

> 注：在所有的信号中，只有SIGKILL/SIGSTOP两个信号不能被捕捉，也不能被忽略。
>
> 只能执行默认的动作。

2.4.5signal函数

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
功能：给进程对信号指定处理方式
参数：
    @signum:信号号
    @handler:处理方式
        SIG_IGN：忽略
        SIG_DFL：默认
        handle:捕捉
        void handle(int signo)
        {

        }
返回值：成功返回handler,失败返回SIG_ERR，置位错误码
```

2.4.6signal函数实例

```
#include <head.h>
void signal_handle(int signo)
{
    if(signo == SIGINT){
        printf("我收到了一个ctrl+c信号\n");
    }
}
int main(int argc, const char* argv[])
{
    // 1.对SIGINT忽略
    // if(SIG_ERR == signal(SIGINT,SIG_IGN))
    //     PRINT_ERR("signal error");

    // 2.对SIGINT默认
    // if(SIG_ERR == signal(SIGINT,SIG_DFL))
    //     PRINT_ERR("signal error");

    // 3.对SIGINT捕捉
    if (SIG_ERR == signal(SIGINT, signal_handle))
        PRINT_ERR("signal error");

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

2.4.7练习

1. 尝试使用signal捕捉管道破裂信号

```c
#include <head.h>
void handle(int signo)
{
    switch (signo) {
    case SIGPIPE:
        printf("我捕捉到了一个管道破裂信号\n");
        break;
    case SIGINT:
        printf("我捕捉到了一个ctrl+c信号\n");
        break;
    }
}
int main(int argc, const char* argv[])
{
    int pipefd[2];
    char s[128] = { 0 };

    if ((SIG_ERR == signal(SIGINT, handle)))
        PRINT_ERR("signal error");

    if ((SIG_ERR == signal(SIGPIPE, handle)))
        PRINT_ERR("signal error");

    if (pipe(pipefd))
        PRINT_ERR("pipe error");

    close(pipefd[0]);

    char ch = 'a';
    while (1) {
        write(pipefd[1], &ch, 1);
        sleep(1);
    }

    return 0;
}
```

2. 尝试使用非阻塞方式回收子进程资源（一定要回收掉子进程资源）

```c
#include <head.h>
char cmd[128] = {0};
void handle(int signo)
{
    waitpid(-1,NULL,WNOHANG);
    printf("我是父进程，我以非阻塞方式回收掉了子进程的资源\n");
    snprintf(cmd,sizeof(cmd),"kill -%d %d",SIGUSR1,getpid());
    system(cmd);
}
int main(int argc, const char* argv[])
{
    pid_t pid;

    if ((pid = fork()) == -1) {
        PRINT_ERR("fork error");
    }else if(pid == 0){
        sleep(7);
        printf("我是子进程，我执行了7s，我现在要退出...\n");
        exit(EXIT_SUCCESS);
    }else{
        if(SIG_ERR == signal(SIGCHLD,handle))
            PRINT_ERR("signal error");
        while(1);
    }
    return 0;
}
```

2.4.8发信号相关函数

```
int raise(int sig);
功能：给自己发信号
参数：
    @sig:信号号
返回值：成功返回0，失败返回非0

int kill(pid_t pid, int sig);
功能：给指定pid的进程发送信号
参数：
    @pid:进程号
```

```
        pid > 0 :给pid号的进程发信号
        pid = 0 :给同组的进程发送信号
        pid = -1:给所有有权限的进程发送信号
        pid <-1:首先会对pid取绝对值，给和这个绝对值相同的组的进程发送信号
    @信号号
返回值：成功返回0，失败返回-1置位错误码

unsigned int alarm(unsigned int seconds);
功能：当seconds倒计时为0的时候发送SIGALRM信号
参数：
    @seconds:秒钟数，如果填写为0，取消挂起的信号
返回值：如果alarm是第一次调用，返回0.
        如果alarm不是第一次调用，返回上一次调用的剩余秒钟数
        alarm(5);  //返回值是0
        sleep(2);  //延时2s
        alarm(5);  //返回值是3
```

2.4.9发信号相关函数实例

raise/kill函数使用实例：

```c
#include <head.h>

void handle(int signo)
{
    waitpid(-1,NULL,WNOHANG);
    printf("我是父进程，我以非阻塞方式回收掉了子进程的资源\n");
    //raise(SIGUSR1);
    kill(getpid(),SIGUSR1);
}
int main(int argc, const char* argv[])
{
    pid_t pid;

    if ((pid = fork()) == -1) {
        PRINT_ERR("fork error");
    }else if(pid == 0){
        sleep(7);
        printf("我是子进程，我执行了7s，我现在要退出...\n");
        exit(EXIT_SUCCESS);
    }else{
        if(SIG_ERR == signal(SIGCHLD,handle))
            PRINT_ERR("signal error");
        while(1);
    }
    return 0;
}
```

alarm的实例1：

```c
#include <head.h>

void handle(int signo)
{
    printf("我收到了闹钟信号\n");
}
int main(int argc,const char * argv[])
{
    if(SIG_ERR == signal(SIGALRM,handle))
        PRINT_ERR("signal error");

    printf("第一次 = %d\n",alarm(5));
    sleep(2);
    printf("第二次 = %d\n",alarm(5));  //SIGALRM

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

alarm的实例2：

```c
#include <head.h>
void handle(int signo)
{
    printf("系统自动出牌了\n");
    alarm(4);
}
int main(int argc,const char * argv[])
{
    if(SIG_ERR == signal(SIGALRM,handle))
        PRINT_ERR("signal error");

    alarm(4);
```

```
    char ch;
    while(1){
        ch = getchar();
        getchar(); //吃'\n'
        printf("用户出牌 = %c\n",ch);
        alarm(4);
    }
    return 0;
}
```