Linux多线程编程 20240222
wuzhh
参考学习视频

阶段一 多线程的简单概念

使用多线程的目的：合理利用资源，提高cpu的效率

- 线程创造，获取ID，生命周期
- 线程控制：终止、连接、取消、发送信号、清除操作
- 线程同步：互斥量、读写锁、条件变量
- 线程高级控制：一次性初始化、线程属性、同步属性、私有数据、安全的fork

1.1 线程

1.1.1 进程与线程

进程：一个正在执行的程序，它是资源分配的最小单位
　　进程中的事情需要按照一定的顺序逐个进行，那么如何让一个进程中的一些事情同时执行 ------> 线程

线程：又称轻量级进程，**程序执行**的最小单位，系统独立调度和分派的基本单位，进程中的一个实体
　　一个进程中可以有多个线程，同时共享进程的资源

> 进程是资源的拥有者，创建/撤销/切换存在较大的时空开销，同时对称多处理机(SMP)可以满足多个运行单位 引入线程；

1.1.2 关于线程的一些术语

- 并发：同一时刻，只能有一条指令执行，但多个进程指令被快速轮换执行，**宏观上**具有多个进程**同时**执行的效果（看起来同时）
- 并行：同一时刻，有多条指令在多个处理器上**同时**执行（真正的同时发送）
- 同步：彼此有依赖关系的调用不应该“同时发生”，而同步就是要**阻止**那些“同时”发送的事情
- 异步：概念和同步相对，任何两个彼此独立的操作都是异步的，表明事件独立的发生

1.1.3 多线程的优势

- 在多处理器中开发程序的并行性
- 在等待慢速IO操作时，程序可执行其他操作，提高并发性
- 模块化的编程，能更清晰的表达程序中独立事件的关系，结构清晰
- 占用较少的系统资源
  　多线程不一定要多处理器

阶段二 Linux线程的基本控制

2.1 创造新线程

2.1.1 线程ID

和进程的PID一样，每个线程也有对应的ID即TID
在线程中，线程ID的类型是`pthread_t`类型，由于在Linux下线程采用POSIX标准，所以，在不同的系统下，`pthread_t`的类型是不同的
ubuntn下，是`unsigned long`类型；solaris系统中，是`unsigned int`类型；在FreeBSD上用的是结构体指针，使用`pthread_equal`来判断

	线程	进程
标识符类型	`pthread_t`	`pid_t`
获取ID	`pthread_self()`	`getpid()`
创建	`pthread_create()`	`fork()`

`pthread_t`：结构体（`FreeBSD5.2、Mac OS10.3`）/`unsigned long int（linux）`/`usr/includebits/pthreadtypes.h`

原文链接：https://blog.csdn.net/weixin_38239856/article/details/83183961

2.1.2 创造线程

函数原型如下：返回值，成功返回0，失败则返回错误码（编译时需要连接库libpthread）

```
int pthread_create(pthread_t *restrict tidp,          //新线程ID，如果成功则新线程的ID会填充到tidp指向的内存
                   const pthread_attr_t *restrict attr,//线程属性，包括调度策略、继承性、分高性、、、
                   void *(*start_routine)(void *),     //回调函数，新线程要执行的函数
                    void *restrict arg)                //回调函数的参数
```

> 错误码一般位置`/usr/include/asm-generic/errno.h`即在**errno.h**文件中

线程创造示例：

```
/* getpid()          获取进程ID
 * pthread_self()     获取线程ID
 * 编译时命令：gcc -lpthread thread_create.c -o thread_create
*/
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
void print_id(char *s)
{
    pid_t pid;
    pthread_t tid;
    pid = getpid();
    tid = pthread_self();
    printf("%s pid is %u, tid is0x%x\n", s, pid, tid);
}

```

```c
void *thread_fun(void *arg)
{
    print_id(arg);
    return(void *)0;
}

int main()
{
    pthread_t ntid;
    int err;
    err = pthread_create(&ntid, NULL, thread_fun, "new thread");
    if(err != 0)        //创建失败
    {
        printf("create new thread failed\n");
        return 0;
    }
    print_id("main thread: ");
    sleep(2);
    return 0;
}
```

2.2 线程的生命周期

2.2.1 初始线程/主线程

- 当c程序运行时，首先运行main函数。在线程代码中，这个特殊的执行流被称为初始线程或主线程
- 主线程的特殊性在于，它在main函数返回的时候，会导致进程结束，进程内所有的线程也会结束
  在主线程中调用`pthread_exit`函数，这样进程就会等待所有线程结束时才终止
- 主线程接收参数的方式时通过`argc`和`argv`，而普通线程只有一个参数`void`
- 绝大多数情况下，主线程在默认堆栈上运行，这个堆栈可以增长到足够的长度。而普通线程的堆栈式是限制的，溢出就会产生错误

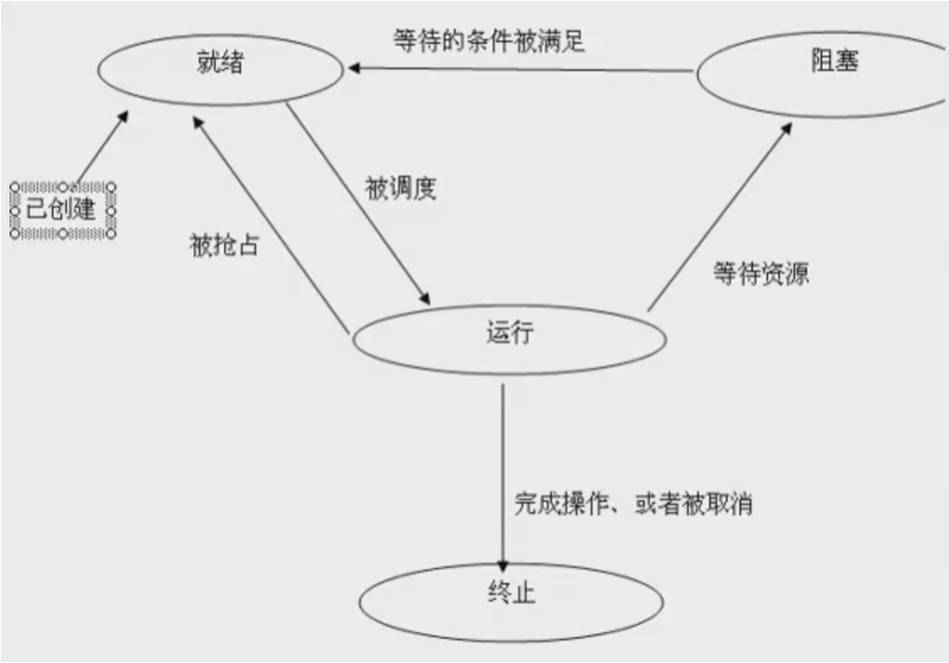
2.2.2 线程的创建

主线程是随着进程的创建而创建（内核完成该操作），其他线程可通过调用函数来创建，主要调用`pthread_create`

新线程可能在当前线程从函数`pthread_create`返回之前运行，甚至新线程可能在当前线程从函数`phread_create`返回之前就运行完毕

2.2.3 线程的四个基本状态

- 就绪：线程能够运行，但在等待可用的处理器
  线程刚被创建时就处于就绪状态，或者当线程解除阻塞以后也会处于就绪状态，就绪的线程在**等待一个可用**的处理器
- 运行：线程正在运行，在多核系统中，可能同时又多个线程在运行；当处理器选中一个就绪的线程执行时，立刻变为运行状态
- 阻塞：线程在等待处理器以外的其他条件；
  阻塞情况：加锁一个已经被锁住的互斥量/等待某个条件变量/调用`singwait`等待尚未发生的信号/执行无法完成的IO信号/内存页错误
- 终止：线程从启动函数中返回，或者调用`pthread_exit`函数，或者被取消



2.2.3 回收

线程的分离属性：

分离一个正在运行的线程并不影响它，仅仅时通知当前系统该线程结束时，其所属的资源可以回收。一个没有被分离的线程在终止时会保留它的虚拟内存，包括他们的堆栈和其他系统资源，也叫这种线程为”**僵死线程**“，创建线程时**默认非分离**

如果线程具有分离属性，线程终止时会被立刻回收，**回收将释放掉所有在线程终止时未释放的系统资源和进程资源**，包括保存线程返回值的内存空间、堆栈、保存寄存器的内存空间等

终止被分离的线程会释放所有的系统资源，但是你**必须释放由该线程占有的程序资源。**由malloc / mmap分配的内存可以在任何时候由任何线程释放，条件变量、互斥量、信号灯可以由任何线程销毁，只要他们被解锁了或者没有线程等待。但是只有互斥量的主人才能解锁它，所以在线程终止前，需要解锁互斥量

2.3 线程的基本控制

2.3.1 线程终止

exit是危险的，如果进程中任意一个线程调用了`exit`，`_Exit`，`_exit`，那么整个进程就会终止

不终止进程的退出方式

- 从启动例程中返回，返回值是线程的退出码
- 线程可以被同一进程中其他线程取消
- 线程调用`pthread_exit(void *rval)`函数，`rval`是退出码

`return`和`pthread_exit`的区别

示例：退出方式的选中
`void pthread_exit(void *rval)` `rval`是个无类型的指针，保存线程的退出码，其他线程可以通过返回码链接这个线程

```
/* 验证几种线程的退出方式
 **/
#include <stdio.h>
#include <pthread.h>
```

```c
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
void *thread_fun(void *arg)
{
    if(strcmp("1", (char *)arg) == 0)        //如果传入参数1，就采用return方式退出
    {
        printf("new thread return!\n");
        return 1;
    }
    if(strcmp("2", (char *)arg) == 0)        //如果传入参数2，那么就采用pthread_exit方式退出
    {
        printf("new thread pthread_exit!\n");
        pthread_exit((void *)2);
    }
    if(strcmp("3", (char *)arg) == 0)        //如果传入参数3，那么就采用exit方式退出
    {
        printf("new thread exit!\n");
        exit(3);
    }
}
int main(int argc, char *argv[])
{
    int err;
    pthread_t tid;
    err = pthread_create(&tid, NULL, thread_fun, (void *)argv[1]);
    if(err != 0)
    {
        printf("create new thread failed\n");
        return 0;
    }
    sleep(1);
    printf("main thread\n");         //采用exit方式退出的话，程序将无法执行此语句，即导致了进程的退出
    return 0;
}
```

2.3.2 线程连接

`int pthread_join(pthread_t tid, void **rval)` 调用成功返回0，失败返回错误码
调用该函数的线程会一直阻塞，直到指定的线程`tid`调用`pthread_exit`从启动例程返回或者被取消
参数`tid`：指定线程的id；参数`rval`：指定线程的返回码，如果线程被取消，那么`rval`被置为`PTHREAD_CANCELED`

调用`pthread_join`会使指定的线程处于**分离状态**，如果指定线程已经处于分离状态，那么调用就会失败

- 用于等待其他线程结束：当调用 pthread_join() 时，当前线程会处于阻塞状态，直到被调用的线程结束后，当前线程才会重新开始执行。
- 对线程的资源进行回收：如果一个线程是非分离的（默认情况下创建的线程都是非分离）并且没有对该线程使用 pthread_join() 的话，该线程结束后并不会释放其内存空间，这会导致该线程变成了“僵尸线程”

`pthread_detach`可以分离一个线程，线程可以自己分离自己

函数`int pthread_detach(pthread_t thread)`成功返回0，失败返回错误码 参数用来指定要分离的线程

```
void *thread_fun1(void *arg)          //新线程启动例程示例函数
{
    printf("I am thread 1\n");        //不做任何操作默认非分离 链接线程调用返回0
    return (void *)1;
}
void *thread_fun2(void *arg)          //线程2
{
    printf("I am thread 2\n");
    pthread_detach(pthread_self());//分离自己 链接线程调用处于分离状态的线程就会失败，返回错误码
    pthread_exit((void *)2);
}
int main()
{
    int err1, err2;
    pthread_t tid1, tid2;
    void *vral1, *vral2;          //创建空变量指针用来传递pthread_join的参数2
    err1 = pthread_create(&tid1, NULL, thread_fun1, NULL);       //线程创建
    err2 = pthread_create(&tid2, NULL, thread_fun2, NULL);
    if(err1 || err2)
    {
        printf("create new thread failed\n");
        return 0;
    }
    printf("I am main thread\n");
    printf("join1 rval is %d\n", pthread_join(tid1, &rval1)); //创造两个回调函数分别连接两个线程 并打印
    printf("join2 rval is %d\n", pthread_join(tid2, &rval2));

    printf("thread1 exit code is %d\n", (int *)rval1);//打印指定线程的退出码 因为rval1定义时是空指针，所在这里要强制转换
    printf("thread2 exit code is %d\n", (int *)rval2); //ling9 调用分离自己函数后无法正确返回退出码
    printf("I am main thread\n");
    return 0;
}
```

2.3.3 线程取消

`int pthread_cancle(pthread_t tid)`参数为：取消tid指定的线程，成功返回0，取消只是发送一个请求，并不意味着等待线程终止，而且发送成功也不意味着tid一定会终止

- 取消函数：`int pthread_cancle(pthread_t tid)`

  参数：取消tid指定线程，成功返回0，取消只是发送一个请求，并不意味等待线程终止，且发送成功也不意味着tid一定会终止
- 取消状态：线程对取消信号的处理方式，忽略或者响应，线程创建时默认响应取消信号

  1、`int pthread_setcancelstate(int state, int *oldstate)`设置本线程对Cancel信号的反应

  2、 state有两种值：`PTHREAD_CANCEL_ENABLE`和`PTHREAD_CANCEL_DISABLE`

  3、分别表示收到信号后设为CANCLED状态和忽略CANCEL信号继续运行；`old_state`如果不为NULL则存入原来的Cancel状态

- 取消类型：线程对取消信号的处理方式，立即取消或者延迟取消，线程创建时默认延时取消
  1、`int pthread_setcanceltype(int type, int *oldtype)` 设置本线程取消动作执行时机
  2、type取值：`PTHREAD_CANCEL_DEFFERED`和`PTHREAD_CANCEL_ASYCHRONOUS`，仅当Cancel状态为Enable时有效
  3、表示收到信号后继续运行至下一个取消点再退出和立即执行取消动作；oldtype如果不为NULL则存入原来的取消动作类型值
- 取消点（查看是否有取消的地方）：取消一个线程，通常需要被取消线程的配合，线程会查看自己是否有取消请求，有就主动退出
  `pthread_join()`,`pthread_testcancel()`,`pthread_cond_wait()`,`pthread_cond_timedwait()`,`sem_wait()`,`sigwait()`,write,read大多数会阻塞系统调用（调用`man pthreads`命令可以查看取消点（Cancellation Points）有哪些）

示例：正确取消一个线

```c
/* 程序框架：主线程创建新线程 -> 新线程将自己的取消状态关闭（不可被取消） -> 新线程休眠，cpu回到主线程 -> 主线程调用    pthread_cancel去给新线程发出一个取消信号，然后继续调用join函数等待新线程的结束 -> CPU回到新线程，线程程打印新的语句然后调用pthread_cancel_enable将自己设为可取消的状态 -> 这时取消信号已经过来，那么新线程就被取消了 -> 新线程取消以后线程就结束了 -> CPU再次回到主线程 -> 主线程打印cancel函数的返回值和新线程的退出码。由于新线程被取消，那么它的退出码则为PTHREAD_CANCEL
 *
*/
void *thread_fun(void *arg)      //新线程启动例程示例函数
{
    int stateval;
    stateval = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    if(stateval != 0)
    {
        printf("set cansel state failed\n");
    }
    printf("I am new thread 1\n");       //不做任何操作默认非分离 链接线程调用返回0
    sleep(4);

    printf("about to cancel\n");
    stateval = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    if(stateval ! = 0)
    {
        printf("set cancel state failed\n");
    }
    printf("first cancel point\n");
    printf("second cancel point\n");         //无法打出
    return (void *)20;
}
int main()
{
    int err, cval, jval;
    int *rval;
    pthread_t tid;

    err = pthread_create(&tid, NULL, thread_fun, NULL);      //线程创建
    if(err1 != 0)
    {
        printf("create new thread failed\n");
        return 0;
    }
    sleep(2);
```

```
    cval = pthread_cancel(tid);
    if(cval != 0)
    {
        printf("cancel thread failed\n");
    }
    jval = pthread_join(tid, &rval);

    printf("New pthread exit code is %d\n", (int *)rval);
    return 0;
}
```

2.3.4 向线程发送信号

(1) `pthread_kill`

`int pthread_kill(pthread_t thread, int sig)` 向线程发送signal大部分signal的默认动作是终止进程的运行，需要用`sigaction()`去抓信号并加上处理函数

向指定ID的线程发送了`SIGQUT`，如果线程代码内不做处理，则按照信号默认的行为影响整个进程，也就是说，如果你给一个线程发送了`SIGQUT`，但线程却没有实现signal处理函数，则整个进程退出

`int sig`参数不为0，需要清除操作目的，且需要实现线程的信号处理函数，否则影响整个进程
参数为0，这是一个保留信号，实际上并没有发送信号，作用是用来判断线程是否还活着

示例：

```
void *thread_fun(void *arg)     //
{
    printf("I am new pthread\n");
    return(void *)0;
}
int main()
{
    pthread_t tid;
    int err;
    int s;
    err = pthread_create(&tid, NULL, thread_cun, NULL);
    if(err != 0)
    {
        printf("create new thread faile\n");
        return;
    }
    sleep(1);
    s = pthread_kill(tid, 0); //若返回错误码为ESRCH，则意味着没找到对应的线程No thread with the ID thread could be found
    if(s == ESRCH)
    {
        printf("thread tid is not found\n");
    }
    return 0;
}
```

（2） 信号处理

进程信号处理：`int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)`

给信号`signum`设置一个处理函数，处理函数在`sigaction`中指定

多线程信号屏蔽处理

```
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
how = SIGBLOCK:向当前的信号掩码中添加set，其中set表示要阻塞的信号组
SIG_UNBLOCK:    向当前的信号掩码中删除set，其中set表示要取消阻塞的信号组
SIG_SETMASK:    将当前的信号掩码替换为set，其中set表示新的信号掩码
/*在多线程中，新线程的当前信号掩码会继承创造它的那个线程的信号掩码*/
/*一般情况下，被阻塞的信号将不能中断此线程的执行，除非此信号的产生是应位程序运行出错如SIGSEGV；另外不能被忽略处理的信号SIGKILL和SIGSTOP也无法被阻塞*/
```

示例：

```

```

2.3.5 清除操作

线程可以安排它退出时的清理操作，这与进程的可以用`atexit`函数安排进程退出时需要调用的函数类似，这样的函数称为线程清理处理函数

线程可以建立多个清理处理程序，**处理程序记录在栈中**，所以这些处理程序执行的顺序与他们注册的顺序相反

`pthread_cleanup_push(void (*rtn)(void*), void *args)` 注册处理程序
`pthread_cleanup_pop(int excute)` 清除处理程序 ----------> 这两个函数要成对出现，否则无法通过编译

执行以下操作时调用清理函数，清理函数的参数由args传入
1 调用`pthread_exit`；2 响应取消请求；3 用非0参数调用`pthread_cleanup_pop`

2.4 线程的同步

2.4.1 互斥量

- 当多个线程共享相同的内存时，需要没有给线程看到相同的视图，当一个线程修改变量时，而其他线程也可以读取或者修改这个变量，就需要对这些线程同步，确保他们不会访问到无效的变量
- 在变量修改时间多于一个存储器访问周期的处理器结构中，当存储器的读和写这两个周期交叉时，这种潜在的不一致性就会出现

```
//多线程访问变量产生错误的例子
/* 输出结果：打印出的三个值不一定相等
 * 原因：通一个结构体，当线程一修改成员变量时，线程二也进行了修改，本身结构体内的成员要求要同步操作，但线程2的加入打破了结构体
 * 操作的原子性
 * 因此在多线程访问变量的时候需要一个同步机制，保证一些操作执行时另外的操作无法执行
*/
struct student{
    int id;
```

```
    int age;
    int name;
}stu;
int i;
void *thread_fun1(void *arg)
{
    while(1)
    {
        stu.id = i;
        stu.age = i;
        stu.name = i;
        i++;
        if(stu.id != stu.age || stu.id != stu.name || stu.age  != stu.name)
        {
            printf("%d, %d, %d\n", stu.id, stu.age, stu.name);
            break;
        }
    }
    return(void *)0;
}
void *thread_fun2(void *arg)
{
    while(1)
    {
        stu.id = i;
        stu.age = i;
        stu.name = i;
        i++;
        if(stu.id != stu.age || stu.id != stu.name || stu.age  != stu.name)
//检查是否有其他线程同时对结构体进行操作
        {
            printf("%d, %d, %d\n", stu.id, stu.age, stu.name);
            break;
        }
    }
    return(void *)0;
}
int main()
{
    pthread_t tid1, tid2;
    int err;
    err = pthread_create(%tid1, NULL, thread_fun1, NULL);
    if(err != 0)
    {
        printf("create new thread failed\n");
        return;
    }
    err = pthread_create(%tid2, NULL, thread_fun2, NULL);
    if(err != 0)
    {
        printf("create new thread failed\n");
        return;
    }
//以确保主线程在继续执行之前等待其他线程完成它们的任务。
//这是一种同步机制，可以防止主线程在子线程完成必要的工作之前继续执行，可能导致竞态条件或其他并发问题。
    //挂起当前线程(通常是主线程)直到线程tid1完成执行。tid1完成pthread_join调用返回，当前线程可继续执行
```

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

（1） 互斥锁的初始化和销毁

为了让线程访问数据不产生冲突，则需要对变量加锁，使得同一时刻只有一个线程可以访问变量
互斥量本质就是**锁**，访问共享资源前对互斥量加锁，访问完成后解锁

当互斥量加锁以后，其他所有要访问该互斥量的线程都将阻塞
当互斥量解锁以后，所有因为这个互斥量阻塞的线程都将变为就绪态，第一个获得CPU的线程会获得互斥量变为**运行态**，而其他线程会继续变为阻塞，这种方式下访问互斥量里每次只有一个线程能向前执行

互斥量用`pthread_mutex_t`类型的数据表示，在使用之前需要对互斥量初始化

1、如果时动态分配的互斥量，可以**调用**`pthread_mutex_init()`**函数**初始化
2、如果时静态分配的互斥量，还可以把它置为常量`PTHREAD_MUTEX_INITIALIZER`
3、动态分配的互斥量在释放内存前需要**调用**`pthread_mutex_destroy()`

```
/* int pthread_mutex_init(pthread_mutex_t *restrict mutex, const
pthread_mutexattr_t *restric attr)
 * 参数1：要初始化的互斥量；      参数2：互斥量的属性，默认为NULL
 * int pthread_mutex_destroy(pthread_mutex_t *mutex);
 * pthrea_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
 * 可在文件路径/usr/include/bits/pthreadtypes.h文件中查看互斥量
 */
```

（2） 加锁和解锁

加锁

`int pthread_mutex_lock(pthread_mutex_t *mutex)`成功返回0，失败返回错误码，如果互斥量已经被锁住，那么会导致该线程阻塞
`int pthread_mutex_trylock(pthread_mutex_t *mutex)`成功返回0，失败返回错误码，如果互斥量被锁住，不会导致线程阻塞

解锁

`int pthread_mutex_unlock(pthread_mutex_t *mutex)`成功返回0，失败返回错误码

```
//互斥锁实例：定义一个结构体，初始成员赋相同的值，有两个线程分别修改它，进行加锁操作（加锁之后修改变量和访问变量变成原子操作）
//正常程序执行结果没有任何输出和返回
struct student{
    int id;
    int age;
    int name;
}stu;
int i;
pthread_mutex_t mutex;  //定义全局互斥量，确保所有线程都可访问到
void *thread_fun1(void *arg)
{
    while(1)
    {
        pthread_mutex_lock(&mutex); //加锁操作 对整个结构体进行加锁 防止产生错乱
```

```
        stu.id = i;
        stu.age = i;
        stu.name = i;
        i++;
        if(stu.id != stu.age || stu.id != stu.name || stu.age  != stu.name)
        {
            printf("%d, %d, %d\n", stu.id, stu.age, stu.name);
            break;
        }
        pthread_mutex_unlock(&mutex);   //解锁操作 加锁和解锁要成对出现 访问变量完成需要进行解锁，方便其他线程访问
    }
    return(void *)0;
}
void *thread_fun2(void *arg)
{
    while(1)
    {
        pthread_mutex_lock(&mutex); //加锁操作
        stu.id = i;
        stu.age = i;
        stu.name = i;
        i++;
        if(stu.id != stu.age || stu.id != stu.name || stu.age  != stu.name) //检查是否有其他线程同时对结构体进行操作
        {
            printf("%d, %d, %d\n", stu.id, stu.age, stu.name);
            break;
        }
        pthread_mutex_unlock(&mutex);   //解锁操作 加锁和解锁要成对出现
    }
    return(void *)0;
}
int main()
{
    pthread_t tid1, tid2;
    int err;
    err = pthread_mutex_init(&mutex, NULL);//主函数初始化一下互斥量，方便使用 只有初始化过的互斥量才能使用
    if(err != 0)
    {
        printf("init mutex failed\n");
        return;
    }

    err = pthread_create(%tid1, NULL, thread_fun1, NULL);
    if(err != 0)
    {
        printf("create new thread1 failed\n");
        return;
    }
    err = pthread_create(%tid2, NULL, thread_fun2, NULL);
    if(err != 0)
    {
        printf("create new thread2 failed\n");
        return;
    }
    //等待新线程运行结束
```

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

（3） 死锁

两个进程独占性的访问某个资源，从而等待另外一个资源的执行结果，会导致两个进程都被阻塞，并且两个进程都不会释放各自的资源，这种情况就是 死锁(deadlock)
如果一组进程中的每个进程都在等待一个事件，而这个事件只能由该组中的另一个进程触发，这种情况会导致死锁

资源死锁：死锁进程结合中的每个进程都在等待另一个死锁进程已经占有的资源。但是由于所有进程都不能运行，它们之中任何一个资源都无法释放资源，所以没有一个进程可以被唤醒。这种死锁也被称为资源死锁(resource deadlock)

资源死锁是最常见的类型，但不是所有的类型

2.4.2 读写锁

- 读写锁和互斥量类似，不过读写锁有更高的并行性，互斥量要么加锁要么不加锁，而且同一时刻只允许一个线程对其加锁，对于一个变量的读取完全可以让多个线程同时进行操作
- `pthread_rwlock_t rwlock` 读写锁有三种状态：读模式加锁、写模式加锁、不加锁
  一次只有一个线程可以占有写模式下的读写锁，但是多个线程可以同时占有读模式的读写锁
- 读写锁在写状态时，在他被解锁之前，所有试图对这个锁加锁的线程都会阻塞
  读写锁在读加锁状态时，所有试图以读模式对其加锁的线程都会获得访问权，但如果线程希望以写模式对其加锁，它必须阻塞直到所有线程释放锁
- 当读写锁读模式加锁时，如果有线程试图以写模式对其加锁，那么读写锁会阻塞随后的读模式锁请求（避免锁长期占用，写不了锁）
- 读写锁适合对数据结构读次数大于写次数的程序，当它以读模式锁住时，是以共享的方式锁住的；当它以写模式写住是，是独占模式锁住

（1） 读写锁的初始化和销毁

读写锁在使用前须初始化 `int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr *restric attr)`

使用完须销毁 `int pthread_rwlock_destroy(pthread_rwlock_t *rwlck)` 成功返回0，失败返回错误码

（2） 加锁和解锁

```
//读模式加锁 Read Mode lock
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock); //试图
//写模式加锁 Write Mode Lock
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
//解锁 unlock
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
/*成功返回0，  失败返回错误码
 * 可使用命令 man pthread_rwlock_init / man pthread_rwlock_wrlock / man pthread_rwlock_trywrlock 查看手册
 * 即man [函数名] 即可查看函数手册*/
```

(3) 读写锁实例

```c
/* 程序流程：读加锁以后线程1访问num，这时线程1休眠5s，CPU执行线程2，如果多个线程可以同时读加锁的话，那么线程2也能访问num变量
 * 并且打印出“thread 2 is over” ，
 * 程序输出几乎同时打印“thread 1 print uum 0”和“thread 2 print uum 0”，并且过10s后
 * 几乎同时打印"thread 1 is over”和"thread 2 is over”
 ***************如果两个线程都使用写加锁的话，程序先输出其中某个线程，然后执行完该线程后才允许其他线程执行*****************
 * 假设程序先输出线程2，那么先打印“thread 2 print uum 0”（说明线程2先执行），5s后（线程1阻塞）输出"thread 2 is over”然后
 * 几乎同时输出“thread 1 print uum 0”（线程1执行），5s后输出"thread 1 is over”
 */
int num = 0;
pthread_rwlock_t rwlock;        //定义一个读写锁 define a read-write lock
void *thread_fun1(void *arg)
{
    int err;
    pthread_rwlock_rdlock(&rwlock); //读加锁  也可替换为写加锁 pthread_rwlock_wrlock(&rwlock);
    printf("thread 1 print uum %d\n", num);
    //以下两行代码用于验证读加锁时多个线程可以使用
    sleep(5);                   //休眠5s
    printf("thread 1 is over\n");
    pthread_rwlock_unlock(&rwlock);//解锁 加锁解锁必须成对出现
    return(void *)1;
}
void *thread_fun2(void *arg)
{
    int err;
    pthread_rwlock_rdlock(&rwlock); //读加锁 也可替换为写加锁 pthread_rwlock_wrlock(&rwlock);
    printf("thread 2 print uum %d\n", num);
    sleep(5);
    printf("thread 2 is over\n");
    pthread_rwlock_unlock(&rwlock);
    return(void *)2;
}
int main()
{
    pthread_t tid1, tid2;
    int err;
    err = pthread_rwlock_init(&rwlock, NULL);   //初始化
    if(err)
    {
        ptintf("init rwlock failed\n");
        return;
    }
    err = pthread_create(%tid1, NULL, thread_fun1, NULL);
    if(err != 0)
    {
        printf("create new thread 1 failed\n");
        return;
    }
    err = pthread_create(%tid2, NULL, thread_fun2, NULL);
    if(err != 0)
```

```
    {
        printf("create new thread 2 failed\n");
        return;
    }
    //等待新线程运行结束
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    pthread_rwlock_destroy(&rwlock);    //销毁
    return 0;
}
```

2.4.3 条件变量

引入实例：
在一条生产线上有一个仓库，当生产者生产的时候需要锁住仓库独占，而消费者取产品时也要锁住仓库独占。如果生产者发现仓库满了，那么他就不能生产了，变成了阻塞状态，但是此时由于生产者独占仓库，消费者又无法进入仓库小号产品，这样就造成僵死状态

需要的机制：当互斥量锁住以后发现当前线程还是无法完成自己的操作，那么它应该释放互斥量，让其他线程工作

方法：

1、采用轮询的方法，不停地查询你需要的条件
2、让系统来帮你查询条件，使用条件变量`pthread_cond_cond`

（1） 条件变量的初始化和销毁

条件变量使用前需要初始化（两种方式）：

1、`pthread_cond_t cond = PTHREAD_COND_INITIALIZER`；静态条件变量
2、`int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr)` 动态申请(malloc)
参数1：要初始化的条件变量 ； 参数2：属性，默认属性为空

条件变量使用完成后需要销毁：
`int pthread_cond_destroy(pthread_cond_t *cond)`

（2） 条件变量的使用

条件变量的使用需要配合互斥量

```
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex)
/*使用pthread_cond_wait等待条件为真，传递给pthread_cond_wait的互斥量对条件进行保护，调用者把锁住的互斥量传递给函数
这个函数将线程放到等待条件的线程列表上，然后对互斥量解锁，这是**原子操作**，当条件满足时函数返回，返回之后继续对互斥量加锁
 */
  int pthread_cond_timedwait(
    pthread_cond_t *restrict cond,
    pthread_mutex_t *restrict mutex,
    const struct timespec *restrict abstime);
/*这个函数与pthread_cond_wait类似，多了timeout，如果到了指定的时间条件还不满足，那么就返回，时间结构体如下*/
struct timespec{
    time_t tv_sec;
    long tv_nsec;
```

```
};
//注意这个时间是绝对时间，例如你要等待3分钟，就要把当前时间加上3分钟然后转换到timespec，而不是直接将3分钟转换到timespec

/*当条件满足时，需要唤醒等待条件的进程 ***** 注意一定要在条件改变之后唤醒线程*/
int pthread_cond_broadcast(pthread_cond_t *cond);//唤醒等待条件的所有线程
int pthread_cond_signal(pthread_cond_t *cond);    //至少唤醒等待条件的某一个线程
```

（3） 条件变量实例

阶段三 linux线程高级控制

3.1 一次性初始化

有些事情需要且只能执行一次，比如互斥量的初始化，通常当初始化应用程序时，可以比较容易地将其放在main函数中，但当你写一个库函数时，就不能在main里面初始化了，你可以用静态初始化，但使用一次初始化（`pthread_once_t`）会比较容易一些

一般步骤：定义一个`pthread_once_t`变量，这个变量要用宏`PTHREAD_ONCE_INIT`初始化。然后创建一个与控制变量有关的初始化函数

```
pthread_once_t = PTHREAD_ONCE_INIT;
void int_routine()
{
    //初始化互斥量
    //初始化读写锁
    /***********/
}
```

接下来就可以在任何时刻调用`pthread_once`函数

`int pthread_once(pthread_once_t *once_control, void(*int_rountine)(void))`
函数功能：该函数使用初值为`PTHREAD_ONCE_INIT`的`once_control`变量保证`init_routine()`函数在本进程执行序列中仅仅执行一次。在多线程编程环境下，尽管`pthread_once()`调用会出现在多个线程中，`init_rountine()`函数仅执行一次，但在哪个线程中执行时不定的，由内核调度来决定

Linux Threads使用互斥锁和条件变量保证由`pthread_once()`指定的函数执行且仅执行一次。
实际上“**一次性函数**”的执行状态有三种：NEVER(0)、IN_PROGRESS(1)、DONE(2)，

用`once_control`表示`pthread_once()`的执行状态：

- `once_control()` **初值为0**，那么`pthread_once`从未执行过，`init_routine()`函数会执行
- `once_control()` **初值为1**，则所有`pthread_once()`都必须等待其中一个激发“已执行一次”信号，因此所有`pthread_once()`都陷入永久等待中（即阻塞），`init_routine()`就无法执行
- `once_control()` **初值为2**，表示`pthread_once()`已执行一次，从而所有`pthread_once()`都立即返回，`init_routine()`就没机会执行
- 当`pthread_once`函数成功返回，`once_control`就会被设置为2

```
//示例：证明pthread_once()只能执行一次
/*程序框架
 * 主线程创建两个新线程 -> 两个新线程分别去调用证明pthread_once函数到底调用了几次
 */
pthread_once_t once = PTHREAD_ONCE_INIT;     //用宏初始化变量 标志pthread_once是否被执行过
pthread_t tid;
```

```c
void thread_init()
{
    printf("I am in thread 0x%x\n", tid);
}
void *thread_fun1(void *arg)
{
    tid = pthread_self();       //获取线程ID
    printf("I am thread 2 0x %x\n", tid);
    pthread_once(%once, thread_init);
    return NULL;
}
void *thread_fun2(void *arg)
{
    sleep(2);       //先睡眠2s 保证线程1先执行
    tid = pthread_self();
    printf("I am thread 1 0x %x\n", tid);
    pthread_once(%once, thread_init);
    return NULL;
}
int main()
{
    pthread_t tid1, tid2;
    int err;
    err = pthread_create(%tid1, NULL, thread_fun1, NULL);
    if(err != 0)
    {
        printf("create new thread 1 failed\n");
        return;
    }
    err = pthread_create(%tid2, NULL, thread_fun2, NULL);
    if(err != 0)
    {
        printf("create new thread 2 failed\n");
        return;
    }
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return 0;
}
```

3.2 线程属性

线程的属性用`pthread_attr_t`类型的结构体表示，在创建线程的时候可以不用传入NULL，而是传入一个`pthread_attr_t`结构，由用户自己来配置线程的属性

`pthread_attr_t`类型对应用程序是不透明的，也就是说应用程序不需要了解有关属性对象内部结构的任何细节，因而可以增加程序的可移植性

线程属性：

名称	描述
`datachstate`	线程的分离属性
`guardsize`	线程栈末尾的警戒区域大小（字节数）
`stacksize`	线程栈的最低地址
`stacksize`	线程栈的大小（字节数）

- 并不是所有的系统都支持线程的这些属性，因此你需要检查当前系统是否支持你设置的属性
- 当然还有一些属性不包含在`pthread_attr_t`结构中，例如：线程的可取消状态、取消类型、并发度

3.2.1 属性的初始化和销毁

`pthread_attr_t`结构在使用之前需要初始化，使用完之后需要销毁

```
int pthread_attr_init(pthread_attr_t *attr);          //线程属性初始化   （分配内存空间）
int pthread_attr_destory(pthread_attr_t *attr);       //线程属性销毁     （释放内存空间）
```

如果在调用`pthread_attr_t`初始化属性的时候**分配了内存空间**，那么`pthread_attr_destory`将释放内存空间，除此之外，`pthread_attr_destory`还会用无效的值初始化`pthread_attr_t`对象，因此该属性对象被误用，会导致创建线程失败

3.2.2 线程的分离属性

（1） 概念

分离一个正在运行的线程并不影响它，仅是通知当前系统该线程结束时，其所属的资源可以回收。一个没有被分离的线程在终止时会保留它的虚拟内存，包括他们的堆栈和其他系统资源，优势这种线程被称为“**僵尸线程**”，创建线程时默认时非分配的

如果线程具有分离属性，线程终止时会被**立刻回收**，回收将释放掉所有在线程终止时未释放的**系统**资源和**进程**资源，包括保存线程返回值的内存空间、堆栈、保存寄存器的内存空间等

（2） 使用方法

修改`pthread_attr_t`结构体的`detachstate`属性，让线程以分离状态启动
使用`pthread_attr_setdetachstate`函数可设置线程的分离状态属性

```
//线程的分离属性有两种合法值
PTHREAD_CREATE_DETACHED      //分离的
PTHREAD_CREATE_JOINABLE      //非分离的，可连接的
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
//设置
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate);
//获取线程的分离状态属性
```

设置线程分离属性的步骤（所有的系统都会支持线程的分离状态属性）

1. 定义线程属性变量`pthread_attr_t attr`
2. 初始化`attr`，`pthread_attr_init(&attr)`

3. 设置线程分为分离或非分离 `pthread_attr_setdetachstate(&attr, detachstate)`
4. 创建线程 `pthread_create(&tid, &attr, thread_fun, NULL)`

(3) 实例

```c
void *thread_fun1(void *arg)
{
    printf("I am thread 2 0x %x\n", tid);
    return(void *)1;
}
void *thread_fun2(void *arg)
{
    printf("I am thread 1 0x %x\n", tid);
    return(void *)2;
}
int main()
{
    pthread_t tid1, tid2;
    int err;

    pthread_attr_t attr;        //定义线程属性变量
    pthread_attr_init(&attr);   //初始化线程属性变量
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);        //设置分离状态属性，置为已分离
    err = pthread_create(%tid1, &attr, thread_fun1, NULL);      //创建线程时确定线程分离属性
    if(err != 0)
    {
        printf("create new thread 1 failed\n");
        return;
    }
    err = pthread_create(%tid2, NULL, thread_fun2, NULL);
    if(err != 0)
    {
        printf("create new thread 2 failed\n");
        return;
    }
    err = pthread_join(tid1, NULL); //已分离的线程无法连接
    if(!err)
        printf("join thread 1 success\n");
    else
        printf("join thread 1 failed\n");
    err = pthread_join(tid2, NULL);
    if(!err)
        printf("join thread 2 success\n");
    else
        printf("join thread 2 failed\n");

    pthread_attr_destroy(&attr);    //销毁属性

    return 0;
}
```

3.2.3 线程栈属性

（1） 线程的栈大小与地址

对进程来说，虚拟地址空间的大小是固定的，进程中只有一个栈，因此它的大小通常不是问题。但对线程来说，同样的虚拟地址被所有的线程共享。如果应用程序使用了太多线程，致使线程栈累计超过可用的虚拟地址空间，这个时候就需要减少线程默认的栈大小。另外，如果线程分配了大量的自动变量或者线程的栈帧太深，那么这个时候需要的栈比默认的大

如果用完了虚拟地址空间，可以使用*malloc*或者*mmap*来为其他栈分配空间，并修改栈的位置

```
int pthread_attr_setstack(pthread_attr_t *attr, void *stackaddr, size_t stacksize);//修改该栈属性
int pthread_attr_getstack(pthread_attr_t *attr, void **stackaddr, size_t *stacksize);//获取栈属性
/* 参数*attr：线程的属性变量
 * 参数stackaddr：栈的内存单元最低地址
 * 参数stacksize：栈的大小
 * 注意stackaddr并不一定是栈的开始，对于一些处理器，栈的地址是从高往低的，那么这是stackaddr是栈的结尾
*/
//单独获取/修改栈的大小（不修改栈的地址），对于栈大小设置，不能小于PTHREAD_STACK_MIN(需要头文件limit.h)
//对于栈大小的设置，在创建线程之后，还可以修改
int pthread_attr_setstacksize(pthread_attr *attr, size_t stacksize);
int pthread_attr_getstacksize(pthread_attr *attr, size_t *stacksize);
```

对于遵循POSIX标准的系统来说，不一定要支持线程的栈属性，需要检查

- 在编译阶段使用
  `_POSIX_THREAD_ATTR_STACKADDR`和`_POSIX_THREAD_ATTR_STACKSIZE`符号来检查系统是否支持线程栈属性
  这些宏定义在`/usr/include/bits/posix_opt.h`文件中
- 在运行阶段使用
  `_SC_THREAD_ATTR_STACKADD`和`_SC_THREAD_ATTR_STACKSIZE`传递给`sysconf`函数检查系统对线程栈属性的支持

（2） 栈尾警戒区

线程属性`guardsize`控制着线程栈末尾以后用以避免栈溢出的扩展内存的大小，这个属性默认是`PAGESIZE`个字节。你可以把它设为0，这样就不会提供警戒缓存区。同样的，如果你修改了`stackaddr`，系统会认为你自己要管理栈，警戒缓冲区会无效

```
int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);  //设置guardsize
int pthread_attr_getguardsize(pthread_attr_t *attr, size_t *guardsize); //获取guardsize
/*不同处理器和不同操作系统对栈的设置不一定相同，所以修改栈属性会降低程序移植性*/
```

（3） 栈属性实例

```
pthread_attr_t attr;          //定义全局变量 线程属性
void *thread_fun(void *arg)
{
    size_t stacksize
```

```
    //获取栈大小 同样需要先检查
#ifdef _POSIX_THREAD_ATTR_STACKSIZE
    pthread_attr_getstacksize(&attr, &stacksize);
    /*
    printf("new thread stack size is %d\n", stacksize);
    pthread_attr_setstacksize(&attr, 18888);    //重新设置 如果超过最小值 那么设置失败栈大小恢复默认值
    pthread_attr_getstacksize(&attr, &stacksize);   //再次获取
    printf("new thread stack size is %d\n", stacksize);//打印新值
    */
#endif
    printf("new thread stack size is %d\n", stacksize);
    return (void *)1;
}
int main()
{
    pthread_t tid;
    int err;
    pthread_attr_init(&attr);       //线程属性初始化，初始化之后才能被使用
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE) //设置分离属性可连接的
    //设置栈的大小 先检查系统是否可以支持栈的设置 检查通过才可以设置
#ifdef _POSIX_THREAD_ATTR_STACKSIZE
    pthread_attr_setstacksize(&attr, PTHREAD_STACK_MIN);
#endif
    err = pthread_create(&tid, &attr, thread_fun, NULL);
    if(err)
    {
        printf("create new thread failed\n");
        return;
    }
    pthread_join(tid, NULL);
    return 0;
}
```

3.3 线程的同步属性

3.3.1 互斥量的属性

类似线程属性一样，线程的同步互斥量也有属性，包括进程共享属性和类型属性。互斥量的属性用`pthread_mutexattr_t`类型的数据表示，在使用之前必须进行初始化，使用完成之后进行销毁

互斥量初始化：`int pthread_mutexattr_init(pthread_mutexattr_t *attr);`
互斥量销毁：`int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);`

（1） 进程共享属性

*进程共享属性需要检测系统是否支持，可以检测宏`_POSIX_THREAD_PROCESS_SHARED`

进程共享属性有两种值

`PTHREAD_PROCESS_PRIVATE`，默认值，同一个进程中的多个线程访问同一个同步对象

`PTHREAD_PROCESS_SHARED`，该属性使互斥量在多个进程中进行同步，如果互斥量在多进程的共享内存区域，那么具有这个属性的互斥量可以同步多进程

设置互斥量进程属性

`int pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict attr, int *restrict pshard);`

`int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);`

(2) 类型属性

互斥量类型	没有解锁时再次加锁	不占用时解锁	已解锁时解锁
`PTHREAD_MUTEX_NORMAL`	死锁	未定义	未定义
`PTHREAD_MUTEX_ERRORCHEK`	返回错误	返回错误	返回错误
`PTHREAD_MUTEX_RECURSIVE`	允许	返回错误	返回错误
`PTHREAD_MUTEX_DEFAULT`	未定义	未定义	未定义

设置互斥量的类型属性

`int pthread_mutexattr_gettype(const pthread_mutexatrr_t *restrict attr, int *restrict type);`

`int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);`

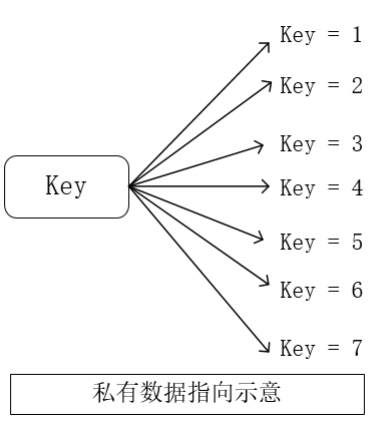
(3) 互斥量属性实例

3.3.2 读写锁的属性

3.3.3 条件变量的属性

3.4 线程私有数据

线程私有数据：多个线程访问这个变量但是对称对这个变量的访问不会产生影响。线程访问的都是这个数据的副本



使用线程私有数据，需要创建一个与私有数据相关的键，用来获取对私有数据的访问权限，键的类型为`pthread_key_t`

`int pthread_key_create(pthread_key_t *key, void(* destructor)(void*));`

创建的键放在key指向的内存单元，destructor是与键相关的析构函数。 当线程调用`pthread_exit`或者使用`return`返回，析构函数就会被调用。当**析构函数**调用的时候，只有一个参数。这个参数是与key关联的那个数据的地址（即私有数据），因此可以在析构函数中将**数据销毁**

键也可以被销毁，但是**销毁键不会销毁键所指向的数据**

`int pthread_key_delete(pthread_key_t key);`

键被创建成功后，将私有数据与键相关联，就可以通过键来找到数据，所有线程都可以访问这个键，但它可以为键关联不同的数据

`int pthread_setspecific(pthread_key_t key,const void* value);`

`void* pthread_getspecific(pthread_key_t key);` 将私有数据与key关联，获取私有数据的地址，如果没有数据与key关联，返回空

```
//example
//两个线程使用同一个函数，来访问同一个数据，但获取到的值是不一样的
pthread_key_t key;  //全局变量 方便两个线程都可访问到
void *thread_fun1(void * arg){
    printf("thread 1 start\n");
    int a = 1;
    pthread_setspecific(key,(void *)a);     //将a和私有数据key关联
    sleep(2);                               //休眠2s cpu转去执行线程2
    printf("thread 1 key->data is %d\n", pthread_getspecific(key));//获取线程1key关联的值 即a = 1
}

void* thread_fun2(void * arg){
    printf("thread 2 start\n");
    int a = 2;
    pthread_setspecific(key,(void *)a);     //关联
    printf("thread 2 key-> data is%d\n",pthread_getspecific(key));//获取关联值
    sleep(1);


}

int main(){
    pthread_t tid1,tid2;
    pthread_key_create(&key, NULL);     //创建一个与私有数据相关的键，虚构函数为空

    if(pthread_create(&tid1,NULL,thread_fun1,NULL))
    {
        printf("create new thread 1 failed\n");
        return;
    }
    if(pthread_create(&tid2,NULL,thread_fun2,NULL))
    {
        printf("create new thread 2 failed\n");
        return;
    }
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    pthread_key_delete(key);        //销毁键

    return 0 ;
}
```

3.5 线程与fork

3.5.1

线程调用fork()函数的时候，子进程创建了整个进程地址空间的副本。子进程通过继承整个地址空间的副本，会将父进程的互斥量、读写锁、条件变量的状态继承过来。如果父进程是锁着的，那么子进程的互斥量也是锁着的，而且这个锁是父进程的，子进程无法解锁

```
//example 安全使用fork() 示例 子进程无法解锁
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;//初始化互斥量
void* thread_fun(void* arg){
    sleep(1);//线程先休眠 主线程先执行
    pid_t pid;
    pid = fork();       //调用fork()函数 产生一个子进程和父进程
    if(pid == 0){
        pthread_mutex_lock(&mutex);//子进程继承了一个加锁的互斥量，所以无法被执行，由于主函数种已锁过一次
        printf("child\n");         //所有这个打印函数无法执行
        pthread_mutex_unlock(&mutex);  //最终会导致该进程一直存在，由于没人给他解锁，所有卡死在这
    }
    if(pid>0){
        pthread_mutex_lock(&mutex); //当运行到父进程时，只要等待主函数解锁后即可加锁
        printf("parent\n");
        pthread_mutex_unlock(&mutex);
    }
}

int main(){
    pthread_t tid;
    if(pthread_create(&tid,NULL,thread_fun,NULL)){
        printf("create new therad failed\n");
        return 0;
    }
    pthread_mutex_lock(&mutex);//互斥量加锁后主线程休眠
    sleep(2);   //主线程休眠两秒 两秒内新线程开始运行
    pthread_mutex_unlock(&mutex);
    printf("main\n");
    pthread_join(tid,NULL);
    return 0;
}
```

3.5.2

子进程内部只有一个线程由父进程中调用fork()函数的线程副本构成。如果调用fork()线程将互斥量锁住，那么子进程会拷贝一个`pthread_mutex_lock`的副本。这样子进程就有机会解锁，但无法确认实际情况

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void* thread_fun(void* arg){
    sleep(1);//主线程先执行
    printf("new thread\n");
    pid_t pid;
    pthread_mutex_lock(&mutex);//在新线程加锁
    pid = fork();
    if(pid == 0){
        pthread_mutex_unlock(&mutex);//可以执行解锁，因为加锁过程在线程中而不是主函数中
```

```
        printf("child\n");
    }
    if(pid>0){
        pthread_mutex_unlock(&mutex);//也可被解锁
        printf("parent process\n");
    }
}

int main(){
    pthread_t tid;
    if(pthread_create(&tid,NULL,thread_fun,NULL)){
        printf("create new therad failed\n");
        return 0;
    }
    printf("main\n");
    pthread_join(tid,NULL);
    return 0;
}
```

3.5.3

```
pthread_atfork();//该函数可以注册三个函数
int pthread_atfork(void(*prepare)(void), void(*parent)(void), void(*child)(void));
```

`void *parepare(void)` 是在fork调用之前会被调用的，parent在fork返回父进程之前调用，child在fork返回子进程之前调用。如果 `parpare` 中加锁所有的互斥量，在 `void* parent(void)`和`(void *) chiled(void)` 中解锁所有的互斥量，那么在fork返回之后，互斥量的状态就是未加锁

可以有多个 `pthread_atfork()` 函数，也需要有多个 `parpare()` 函数， `parpare()` **函数的执行顺序与注册顺序相反，parent与child的执行顺序与注册顺序相同**

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void parpare(){     //fork调用之前调用的 目的是将锁的状态获取
    pthread_mutex_lock(&mutex);
    printf("parpare is running\n");
}
void parent(){//在fork返回父进程之前调用，目的为解锁
    pthread_mutex_unlock(&mutex);
    printf(" parent \n");
}
void child(){   //返回子进程之前调用，目的为解锁
    pthread_mutex_unlock(&mutex);
    printf("child\n");
}

void* thread_fun(void* arg){
    sleep(1);//主线程先执行
    pid_t pid;
    pthread_atfork(parpare,parent,child);       //注册三个函数
    pid = fork();
    if(pid == 0){//子进程继承了一个加锁的互斥量，所以无法被执行
        pthread_mutex_lock(&mutex);
        printf("child process\n");
        pthread_mutex_unlock(&mutex);
    }
```

```
    if(pid>0){
        pthread_mutex_lock(&mutex);
        printf("parent process\n");
        pthread_mutex_unlock(&mutex);
    }
}

int main(){
    pthread_t tid;
    if(pthread_create(&tid,NULL,thread_fun,NULL)){
        printf("create new therad failed\n");
        return 0;
    }
    pthread_mutex_lock(&mutex);//加锁后主线程休眠
    sleep(2);
    pthread_mutex_unlock(&mutex);
    printf("main\n");
    pthread_join(tid,NULL);
    return 0;
}
```

阶段四 linux多线程综合联系

4.1 基于多线程的TCP并发服务器

TCP服务器创建的步骤

4.1.1 步骤一

创建一个socket，使用函数socket()

- Socket（套接字）实际上提供来进程通信的断电，进程通信之前，双方首先必须建立各自的一个端点，否者无法通信。通过socket将IP地址和端口绑定之后，客户端就可以和服务器通信
- 访问套接字时，更像访问文件一样使用文件描述符
- 使用socket()函数可创造一个套接字：`int socket(int domain, int type, int protocol)` 成功返回套接字文件描述符，失败返回-1
    a. **参数domain**：通信域，确定通信特征，包括地址格式

域	描述
`AF_INET`	IPv4因特网域
`AF_INET6`	IPv6因特网域
`AF_UNIX`	unix域
`AF_UNSPEC`	未指定

b. **参数type**：套接字类型

类型	描述
SOCK_DGRAM	长度固定的、无连接的不可靠报文传输
SOCK_RAW	IP协议的数据报接口
SOCK_SEQPACKET	长度固定、有序、可靠的面向连接报文传递
SOCK_STREAM	有序、可靠、双向的面向连接的字节流

c. **参数protocol**：指定相应的传输协议，也就是诸如TC P或UDP协议等等，系统针对每一个协议族与类型提供了一个默认的协议，完美通过把protocol设置为0来使用这个默认的值

4.1.2 步骤二

绑定IP地址和端口信息到socket，使用函数bind()

（1） IP地址

在socket程序设计中，`struct sockaddr_in`（或者`struct sock_addr`）用于记录网络地址

```
struct sockaddr_in{
    short int sin_family;//协议族  和socket中的通信域相匹配
    unsigned short int sin_port;//端口号
    struct in_addr sin_addr;//协议特定地址
    unsigned char sin_zero[8];//填0
};
typedef struct in_addr{
    union{
        struct{
            unsigned char s_b1,
            s_b2,
            s_b3,
            s_b4;
        }S_un_b;
        struct{
            unsigned short s_w1,
            s_w2;
        }S_un_w;
        unsigned long s_addr;
    }S_un;
}IN_ADDR
```

IP地址通常由数字加点（192.168.0.1）的形式表示，而在`struct in_addr`中使用的IP地址是有32位的整数表示的，利用以下函数转换

```
int inet_aton(const char* cp,struct in_addr *inp);//inet_aton 将a.b.c.d 转换成32bit的IP，存储在inp指针里面。
char *inet_ntoa(struct in_addr in);//inet_ntoa 将32bitIP转换成a.b.c.d的格式
// 函数中的a 代表ascii , n 代表network
```

不同类型的的CPU对变量的字节存储顺序可能不同，大端或者小端。但网络传输的数据顺序是统一的，当内部字节存储舒徐和网络字节顺序不同时，需要进行转换

hton -->`unsigned short`类型	**从host（主机序）转换成network（网络序）**
htonl -->`unsigned long`类型	**从host（主机序）转换成network（网络序）**
ntohs -->`unsigned short`类型	**从network（网络序）转换成host（主机序）**
ntohl -->`unsigned long`类型	**从network（网络序）转换成host（主机序）**

（2） bind()

绑定服务器地址和端口到socket，这样做为了让客户端来发现用以连接的服务器地址

```
int bind(int sockfd,const struct sockaddr * addr,socklen_t len);
/* 返回值：成功返回0,失败返回-1
 * 参数sockfd：服务器socket
 * 参数addr： 服务器的地址，设置为INADDR_ANY套接字可以绑定到所有网络端口。一般使用sockaddr_in类型代替sockaddr类型
 * 参数len：addr的长度
 */
```

4.1.3 步骤三

设置允许的最大连接数，使用函数listen()

listen() 服务器调用listen函数来宣告可以接受连接请求

`int listen(int sockfd,int backlog);`
返回值：成功返回0,失败返回-1。
参数`backlog`：是服务器能接受的请求数量

4.1.4 等待来自客户端的连接请求，使用函数accept()

服务器调用listen后套接字就能接收连接请求，使用函数`accept()`来接受并建立请求

```
int accept(int sockfd,struct sockaddr *restrict addr, socklen_t* restrict len);
/* 返回值：成功返回1,失败返回 -1
 * 参数sockfd：服务器socket
 * 参数addr：客户端的地址
 * 参数len: addr的长度
*************************       注意       ****************
**1、accept返回一个新的socket关联到客户端，与原始的socket有相同的套接字类型和协议族。传递给accpet的原始socket并没有关联客户端，要继续保持原有的状态，接受其他请求。（同时可能有多个客户端进行请求通信）
**2、accept()是阻塞的函数，会一直等待客户端的请求
 */
```

4.1.5 步骤四

收发数据，用函数`recv()、send()/sendto()`或者`read()、write()`

4.1.6 步骤五

关闭网络连接，close

4.1.7 TCP客户端的创建

1. 创建一个socket，使用函数socket()
2. 设置要连接的服务器地址和端口
3. 连接服务器，使用函数connect()
4. 收发数据，用函数 `recv()、send()/sendto()` 或者 `read()、write()`
5. 关闭网络连接，close

4.2 实例

```
/*客户端域服务器之间的数据交换时双向的，互不影响，因此需要两个单独的线程来读和写*/
/*简易TCPf*/
//服务器
#define MAXLINE 7890
int main(int argc, char** argv)
{
    int    listenfd, connfd;
    struct sockaddr_in     servaddr;
    char    buff[4096];
    int     n;

    if( (listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 ){
    printf("create socket error: %s(errno: %d)\n",strerror(errno),errno);
    return 0;
    }

    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;      //协议族
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);//主机地址 利用宏来自动匹配网卡读取地址
    servaddr.sin_port = htons(6789);        //端口

    if( bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) == -1){
    printf("bind socket error: %s(errno: %d)\n",strerror(errno),errno);
    return 0;
    }

    if( listen(listenfd, 10) == -1){
    printf("listen socket error: %s(errno: %d)\n",strerror(errno),errno);
    return 0;
    }

    printf("======waiting for client's request======\n");
   // while(1){
    if( (connfd = accept(listenfd, (struct sockaddr*)NULL, NULL)) == -1){
        printf("accept socket error: %s(errno: %d)",strerror(errno),errno);
       // continue;
    }
    n = recv(connfd, buff, MAXLINE, 0);
    buff[n] = '\0';
    printf("recv msg from client: %s\n", buff);
```

```
    close(connfd);
  //  }

    close(listenfd);
}
/*****************************************************************************
*************************************/
//客户端
#define MAXLINE 4096
int main(int argc, char** argv)
{
    int    sockfd, n;
    char    recvline[4096], sendline[4096];
    struct sockaddr_in    servaddr;

    if( argc != 2){
    printf("usage: ./client <ipaddress>\n");
    }

    if( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0){
    printf("create socket error: %s(errno: %d)\n", strerror(errno),errno);
    }

    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(5658);
    if( inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0){
    printf("inet_pton error for %s\n",argv[1]);
    }

    if( connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) < 0){
    printf("connect error: %s(errno: %d)\n",strerror(errno),errno);
    }

    printf("send msg to server: \n");
    fgets(sendline, 4096, stdin);
    if( send(sockfd, sendline, strlen(sendline), 0) < 0){
    printf("send msg error: %s(errno: %d)\n", strerror(errno), errno);
    }
    close(sockfd);
}
```