线程

1. 线程标识;

线程id是用pthread_t数据类型来表示的

- 对两个线程ID进行比较

```
#include <pthread.h>
int phread_equal(pthread_t tid1,pthread_t tid2);
返回值：若相等返回非0，否则返回0
```

- 因为pthread_t 数据类型在不同的系统版本中代表的数据类型有差异，所以不能用一种可移植的方式来打印线程ID，所以线程可以通过下面的函数来获得自身的线程ID

```
pthread_t pthread_self(void)
```

- **线程的一个引用场景：主线程把新的作业放到一个工作队列中，由3个工作线程组成的线程池从队列中移出工作，主线程不允许每个线程任意处理从队列顶端取出的作业，而是由主线程控制作业的分配。主线程会在每个待处理作业的结构中放置处理该作业的id。每个工作线程只能移出标有自己线程ID的作业**
-

2. 线程创建

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

3. 线程概念

- 进程调用pthread_create()之后就变成了线程，
- 进程：有独立的进程地址空间，有独立的pcb

  线程：有独立的pcb，没有独立的进程地址空间。

  区别：是否共享地址空间

  - LINUX下：线程：最小的执行单位

    进程：最小分配资源单位，可以看成时只有一个线程的进程

```
查看进程中的线程 ps -Lf pid
线程号 LWP
```

4. 线程之间贡献资源

- 文件描述表

- 每种信号的处理方式
- 当前工作目录
- 用户ID和组ID
- 内存地址空间(没有栈)

5. 线程之间非共享资源
    - 线程ID
    - 处理器现场和栈指针（内核栈）
    - 独立的栈空间(用户空间栈)
    - errno变量（本质是个全局变量）
    - 信号屏蔽字（mask）
    - 调度优先级

6. 线程控制原语

- pthread_self函数

    获取线程ID 其作用对应进程中getpid()函数

```
pthread_t pthread_self(void)
线程ID pthread类型
线程ID是进程内部，识别标志（两个进程间，线程ID允许相同）
注意：不应使用全局变量pthread_t tid,在子线程中通过pthread_create传出参数来获取
     线程ID.二应使用pthread_self
```

- pthread_create函数

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                           void *(*start_routine) (void *), void *arg);
pthread_t *thread：传出参数，创建成功的线程ID
pthread_attr_t *attr：线程属性（优先级等），没有特殊需求可以传NULL
void *(*start_routine)：线程执行逻辑的函数
void *arg：线程函数的参数
编译的时候 参数
Compile and link with -pthread.
```

    demo

```
 7 void sys_err(const char *str)
 8 {
 9         perror(str);
10         exit(0);
11 }
12
13 void *math(void *arg)
14 {
```

```
15          int i = arg;
16          printf("num is %d,%d \n",i,arg);
17  }
18  int main(void)
19  {
20          int i;
21          pthread_t tid;
22          for(i = 0; i < 5; i++)
23          {
24                  printf("------------%d-------------\n", (int*) i);
25          pthread_create(&tid,NULL,math,(void*)i);//传参采用 值传递，借助
                                                    强转，不能传地址这里借助的是地址的值
26          }
27          sleep(10);
28  return 0;
29  }
```

-
-
-

7. 主线程和子线程是共享全局变量的
8. pthread_exit()函数

```
void pthread_exit(void *retval);退出当前线程
```

- demo

```
pthread_exit(NULL);
```

-
-
-

9. pthread_join函数

- 回收线程

```
int pthread_join(pthread_t thread, void **retval);
pthread：回收的进程ID
**retval：代表线程的退出状态,传出参数,子线程退出时的值
成功返回0 失败返回错误号

补充
```

```
比如进程退出的值为0（int 类型），wait回收的值为int *类型
线程退出值为void* 类型，回收值值为void**
```

-

10. pthread_cancel函数：杀死线程

```
int pthread_cancel(pthread_t thread);
thread：传出参数
```

11. pthread_detach函数

```
int pthread_detach(pthread_t thread);
thread：要分离的线程ID
```

12. 线程属性：

```

```

13. 线程使用注意事项

- 主线程退出其他线程不退出，主线程应该调用pthread_exit
- 避免僵尸进程：pthread_join
- malloc和mmap申请的内存可以被其他线程释放
- 应避免在多线程模型中调用fork除非马上exec。
- 避免在多线程引入信号机制

14. 线程同步

同步即同步调，按预定的先后次序运行。

线程同步，指一个线程发出某一功能调用时，在没有得到结果之前，该调用不返回。同时其它线程为保证数据一致性，不能调用该功能

- 数据混乱原因

  资源共享

  调度随机

  线程间缺乏必要的同步机制

15. 互斥量（互斥锁）mutex

所有线程应该在访问公共数据前先拿锁再访问，但锁不具备强制性

- 主要应用函数

pthread_mutex_init()：初始化一个互斥锁

```
pthread_mutex_init(pthread_mutex_t *restrict mutex,
            const pthread_mutexattr_t *restrict attr);
pthread_mutex_t *restrict mutex：restrict是一个关键字，表示本指针指向地址空间的内容操作，由本
const pthread_mutexattr_t *restrict attr：属性
```

pthread_mutex_lock():

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

```
流程：
pthread_mutex_t lock;创建锁；可以想象成一个整数
pthread_mutex_init;初始化，可以认为初始化后锁的值为1
pthread_mutex_lock;加锁 可以想象成锁--  （1-- = 0）//拿锁，如果这个锁被其他线程拿走就会造成阻塞
                              try锁：尝试加锁，成功（--），失败（返回错误号EBUSY）
访问共享数据
pthread_mutext_unlock();解锁可以想象成++（变为1）
Pthread_mutex_destroy()；销毁锁

```

16. 读写锁

```
 https://blog.csdn.net/dangzhangjing97/article/details/80368822
```

17. 死锁

- 对一个锁反复lock
- 两个线程各自拥有一把锁，各自请求另一把锁

18. 条件变量：等待条件满足

本身不是锁，到那时需要结合锁来使用

```
主要应用函数:
pthread_cond_destroy():初始化一个条件变量，用完销毁
pthread_cond_wait():等待一个条件满足：先要创建一个互斥锁，参数需要
                    1、阻塞等待条件变量（参数1）满足；2、释放已掌握的互斥锁（参数2）pthread_mut
                    3、当被唤醒，pthread_cond_wait函数返回时。会重新上锁（参数2的锁）
                    等待条件变量之前需要已经有几个操作完成
                    pthread_mutex_t m;
                    pthread_mutex_init()
                    pthread_mutex_lock()
pthread_cond_signal():唤醒阻塞在条件变量上的一个线程
Pthread_cond_broadcast():唤醒阻塞在条件变量上的所有线程
pthread_cond_timewait():等待阻塞的时间，如果时间达到就返回不再阻塞
```

- 条件变量的生产者和消费者模型

```
1
```

-
-
-

19.
20.
21.