作者：尹德位　　http://my.oschina.net/cnyinlinux　　cnyinlinux@163.com

# POSIX 线程库功能接口与知识点汇总

## —— Linux 平台

2016 元月 西安

库　　：　/lib64/libpthread.so*
头文件：　pthread.h

目　录



---

开　篇

本文只是针对 POSIX 线程库功能接口与知识点的汇总罗列，仅供多线程编程时查阅参考。

# 第一篇 线程创建与控制

1.线程标识符 pthread_t
此类型定义如下
/usr/include/bits/pthreadtypes.h : typedef unsigned long int pthread_t ;

注：线程有两重身份：1).线程身份;　2).子进程身份(各线程也具有 pid)

2.线程创建

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

返回值：0 成功
参数：thread 线程 ID 变量地址，　attr 线程属性结构地址
　　start_routine 线程入口函数，　arg 线程入口函数的参数列表

3.线程终止退出函数

```
void pthread_exit(void *retval);
```

入参为退出状态码保存地址，一般无视传入 NULL

4.线程回收函数

```
int pthread_join(pthread_t thread, void **retval);
```

父线程回收子线程时调用，若子线程未退出则挂起等待。
入参为目标线程的 ID 和退出状态，若目标线程被取消，则 PTHREAD_CANCELED 保存在*retval

5.线程分离

```
int pthread_detach(pthread_t thread);
```

将子线程分离出去，子线程结束后系统自动回收资源

分离后的子线程不可用 pthread_join 回收(报错返回)

6.线程信号发送

int pthread_kill(pthread_t thread, int sig);

注：线程与进程信号原理一样，但影响范围不同。不可采用进程的信号处理方式对待线程
单一线程收到进程级信号，则影响整个进程

7.线程取消

int pthread_cancel(pthread_t thread);

取消一个线程即终止它，但不是立即结束，而是到下一个取消点到达时
取消点通常位于系统函数中。
人工取消点：void pthread_testcancel(void);
若线程中不存在具有取消点的函数调用，则人工加入取消点函数即可。

8.线程信号屏蔽字

int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);

此用法与进程信号屏蔽字设置函数 sigprocmask 相似，仅作用范围不同

9.线程 ID 比较函数

int pthread_equal(pthread_t t1, pthread_t t2);

判断两个线程 ID 是否相等

10.获取线程自身 ID

pthread_t pthread_self(void);

返回值为自线程的 ID

# 第二篇 线程属性设置

1.线程属性结构 pthread_attr_t

线程大多属性都可通过该结构体设置，在创建线程时通过 pthread_create 的第二个参数使其生效。其定义如下

/usr/include/bits/pthreadtypes.h : typedef union pthread_attr_t pthread_attr_t;

```
/*----------------------------------------------------------*/
# ifdef __x86_64__
# if __WORDSIZE == 64
#    define __SIZEOF_PTHREAD_ATTR_T 56
# else
#    define __SIZEOF_PTHREAD_ATTR_T 32
# endif
# else
# define __SIZEOF_PTHREAD_ATTR_T 36
# endif
union pthread_attr_t
{
    char __size[__SIZEOF_PTHREAD_ATTR_T];
    long int __align;
};
/*----------------------------------------------------------*/
```

2.初始化/销毁 线程属性结构体

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

3.设置/获取 线程分离属性

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *detachstate);
```

线程分离属性值仅有 2 个可选项

PTHREAD_CREATE_DETACHED 分离

PTHREAD_CREATE_JOINABLE 未分离

4.设置/获取 线程栈信息

```
int pthread_attr_setstack(pthread_attr_t *attr,
                          void *stackaddr, size_t stacksize);
int pthread_attr_getstack(pthread_attr_t *attr,
                          void **stackaddr, size_t *stacksize);
```

若指定栈空间，栈大小必须不小于 PTHREAD_STACK_MIN

还有两组关于栈信息的古老函数，建议弃用：

```
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);      //已过时，弃用
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);     //已过时，弃用
```

int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr); //已过时，弃用
int pthread_attr_getstackaddr(pthread_attr_t *attr, void **stackaddr); //已过时，弃用

5.设置/获取 栈警戒区大小
int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
int pthread_attr_getguardsize(pthread_attr_t *attr, size_t *guardsize);
防止栈溢出而设置的警戒区，应大于 0

6.设置取消状态
int pthread_setcancelstate(int state, int *oldstate);
取消状态有 2 个可选项
PTHREAD_CANCEL_ENABLE 可取消
PTHREAD_CANCEL_DISABLE 不可取消

7.设置取消类型
int pthread_setcanceltype(int type, int *oldtype);
取消类型有 2 个可选项
PTHREAD_CANCEL_DEFERRED 取消响应在下次取消点到达时
PTHREAD_CANCEL_ASYNCHRONOUS 取消响应为任意时刻点

# 第三篇 线程同步技术

1.线程锁(互斥量) pthread_mutext_t

定义在/usr/include/bits/pthreadtypes.h 中

```
/*----------------------------------------------------------*/
typedef union
{
    struct __pthread_mutex_s
    {
      int __lock;
      unsigned int __count;
      int __owner;
#ifdef __x86_64__
      unsigned int __nusers;
#endif
      /* KIND must stay at this position in the structure to maintain
         binary compatibility.  */
      int __kind;
#ifdef __x86_64__
      int __spins;
      __pthread_list_t __list;
# define __PTHREAD_MUTEX_HAVE_PREV 1
#else
      unsigned int __nusers;
      __extension__ union
      {
        int __spins;
        __pthread_slist_t __list;
      };
#endif
    } __data;
    char __size[__SIZEOF_PTHREAD_MUTEX_T];
    long int __align;
} pthread_mutex_t;
/*----------------------------------------------------------*/
```

1-0.互斥量的种类：

A. PTHREAD_MUTEX_TIMED_NP

B. PTHREAD_MUTEX_RECURSIVE_NP

C. PTHREAD_MUTEX_ERRORCHECK_NP

D. PTHREAD_MUTEX_ADAPTIVE_NP

E. PTHREAD_MUTEX_FAST_NP

| 锁类型 | 初始化方式 | 加解锁特征 | 调度特征 |
| --- | --- | --- | --- |
| 普通锁 | PTHREAD_MUTEX_INITIALIZER | 加锁一次<br>解锁一次 | 先等待<br>先获得 |
| 嵌套锁 | PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP | 同一线程可重复加锁，解锁同样次数才可释放锁 | 自由竞争 |
| 纠错锁 | PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP | 重复加锁报错，只能由本线程解锁 | 先等待先获得 |
| 自适应锁 | PTHREAD_ADAPTIVE_MUTEX_INITIALIZER_NP | 加锁一次<br>解锁一次 | 自由竞争 |

1-1.锁的初始化/销毁

pthread_mutex_t mutex;

1). 静态方式：mutex = PTHREAD_MUTEX_INITIALIZER ;

2). 动态方式：int pthread_mutex_init(pthread_mutex_t *restrict mutex,
const pthread_mutexattr_t *restrict attr);
^锁的种类由 attr 结构指定

3).锁的销毁 ： int pthread_mutex_destroy(pthread_mutex_t *mutex);

1-2.锁的属性设置 pthread_mutexattr_t

A.初始化/销毁 锁属性结构

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

B.设置/获取 锁属性(种类)

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

type 可选项：

```
PTHREAD_MUTEX_NORMAL
PTHREAD_MUTEX_ERRORCHECK
PTHREAD_MUTEX_RECURSIVE
PTHREAD_MUTEX_DEFAULT
PTHREAD_MUTEX_FAST_NP
```

1-3.加锁/解锁

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
                            const struct timespec *restrict abs_timeout);
```

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

2.自旋锁 pthread_spinlock_t
与互斥锁类似，但不释放 CPU，适用于短程占用高频抢占式场景
2-0.自旋锁初始化/销毁

```
int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
int pthread_spin_destroy(pthread_spinlock_t *lock);
```

注：pshared 可选项为：

```
PTHREAD_PROCESS_PRIVATE
PTHREAD_PROCESS_SHARED
```

2-1.自旋锁加解锁

```
int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

3.读写锁 pthread_rwlock_t
用于读多写少场景
3-0.读写锁初始化/销毁

```
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                        const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

3-1.读写锁属性 pthread_rwlockattr_t
初始化/销毁

```
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

设置/获取 属性

```
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *
                                  restrict attr, int *restrict pshared);
```

pshared 可选项为：

```
PTHREAD_PROCESS_PRIVATE
PTHREAD_PROCESS_SHARED
```

3-2.读写锁加解锁

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);        //读加锁
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);        //写加锁
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);        //解锁(读/写 通吃)
```

4.条件变量 pthread_cond_t
事件触发机制技术，与线程锁协同使用

4-0.条件变量初始化/销毁

```
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

4-1.条件变量属性 pthread_condattr_t

初始化/销毁

```
int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);
```

设置/获取 属性

```
int pthread_condattr_setpshared(pthread_condattr_t *attr, int pshared);
int pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,
                                int *restrict pshared);
```

pshared 可选项为：

PTHREAD_PROCESS_PRIVATE

PTHREAD_PROCESS_SHARED

4-2.等待条件变量

与线程锁协同工作，wait 之前应确保线程锁成功加锁，

下述 wait 函数内部率先解锁而后等待条件变化(挂起)，函数返回前再次加锁。

```
int pthread_cond_wait(pthread_cond_t *restrict cond,pthread_mutex_t *restrict mutex);
int pthread_cond_timedwait(pthread_cond_t *restrict cond,pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict abstime);
```

使用流程：

```
/*-------------------------------------------------------------*/
pthread_mutex_lock(&mutex);
pthread_cond_wait(&cond,&mutex);  //内部先解锁，而后等条件，最后加锁后返回
pthread_mutex_unlock(&mutex);
/*-------------------------------------------------------------*/
```

4-3.唤醒条件挂起线程

当其他线程完成工作释放条件时，唤醒其他线程

但条件变量是互斥使用，只可能有一个线程获得该条件控制权

```
int pthread_cond_signal(pthread_cond_t *cond);        //只唤醒其中一个线程
int pthread_cond_broadcast(pthread_cond_t *cond);     //可能会唤醒所有线程，产生惊群现象
```

<<END>>