第九课　线程管理

一、基本概念

1. 线程就是程序的执行路线，即进程内部的控制序列，
   或者说是进程的子任务。

2. 线程，轻量级，不拥有自己独立的内存资源，
   共享进程的代码区、数据区、堆区(注意没有栈区)、
   环境变量和命令行参数、文件描述符、信号处理函数、
   当前目录、用户ID和组ID等资源。

3. 线程拥有自己独立的栈，因此也有自己独立的局部变量。

4. 一个进程可以同时拥有多个线程，
   即同时被系统调度的多条执行路线，
   但至少要有一个主线程。

二、基本特点

1. 线程是进程的一个实体，
   可作为系统独立调度和分派的基本单位。

2. 线程有不同的状态，系统提供了多种线程控制原语，
   如创建线程、销毁线程等等。

3. 线程不拥有自己的资源，只拥有从属于进程的全部资源，
   所有的资源分配都是面向进程的。

4. 一个进程中可以有多个线程并发地运行。
   它们可以执行相同的代码，也可以执行不同的代码。

5. 同一个进程的多个线程都在同一个地址空间内活动，
   因此相对于进程，线程的系统开销小，任务切换快。

6. 线程间的数据交换不需要依赖于类似IPC的特殊通信机制，
   简单而高效。

7. 每个线程拥有自己独立的线程ID、寄存器信息、函数栈、
   错误码和信号掩码。

8. 线程之间存在优先级的差异。

三、POSIX线程(pthread)

1. 早期厂商各自提供私有的线程库版本，
   接口和实现的差异非常大，不易于移植。

2. IEEE POSIX 1003.1c (1995)标准，
   定义了统一的线程编程接口，
   遵循该标准的线程实现被统称为POSIX线程，即pthread。

3. pthread包含一个头文件pthread.h，
   和一个接口库libpthread.so。

```
#include <pthread.h>

...

gcc ... -lpthread
```

4. 功能

1) 线程管理：创建/销毁线程、分离/联合线程、
   设置/查询线程属性。

2) 线程同步

A. 互斥量：创建/销毁互斥量、加锁/解锁互斥量、
   设置/查询互斥量属性。

B. 条件变量：创建/销毁条件变量、等待/触发条件变量、
   设置/查询条件变量属性。

四、线程函数

1. 创建线程

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

thread - 线程ID，输出参数。
  pthread_t即unsigned long int。

attr - 线程属性，NULL表示缺省属性。
  pthread_attr_t可能是整型也可能是结构，
  因实现而异。

start_routine - 线程过程函数指针，
  参数和返回值的类型都是void*。
  启动线程本质上就是调用一个函数，
  只不过是在一个独立的线程中调用的，
  函数返回即线程结束。

arg - 传递给线程过程函数的参数。
  线程过程函数的调用者是系统内核，
  而非用户代码，
  因此需要在创建线程时指定参数。

成功返回0，失败返回错误码。

注意：

1) restrict: C99引入的编译优化指示符，
   提高重复解引用同一个指针的效率。

2) 在pthread.h头文件中声明的函数，
   通常以直接返回错误码的方式表示失败，
   而非以错误码设置errno并返回-1。

3) main函数即主线程，main函数返回即主线程结束，
   主线程结束即进程结束，
   进程一但结束其所有的线程即结束。

4) 应设法保证在线程过程函数执行期间，
   其参数所指向的目标持久有效。

创建线程。范例：create.c

线程并发。范例：concur.c

线程参数。范例：arg.c

2. 等待线程

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

等待thread参数所标识的线程结束，
成功返回0，失败返回错误码。

范例：ret.c

注意从线程过程函数中返回值的方法：

1) 线程过程函数将所需返回的内容放在一块内存中，
   返回该内存的地址，保证这块内存在函数返回，
   即线程结束，以后依然有效；

2) 若retval参数非NULL，
   则pthread_join函数将线程过程函数所返回的指针，
   拷贝到该参数所指向的内存中；

3) 若线程过程函数所返回的指针指向动态分配的内存，
   则还需保证在用过该内存之后释放之。

3. 获取线程自身的ID

```
pthread_t pthread_self (void);
```

成功返回调用线程的ID，不会失败。

4. 比较两个线程的ID

```
int pthread_equal (pthread_t t1, pthread_t t2);
```

若参数t1和t2所标识的线程ID相等，则返回非零，否则返回0。

某些实现的pthread_t不是unsigned long int类型，
可能是结构体类型，无法通过“==”判断其相等性。

范例：equal.c

5. 终止线程

1) 从线程过程函数中return。

2) 调用pthread_exit函数。

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

retval - 和线程过程函数的返回值语义相同。

注意：在任何线程中调用exit函数都将终止整个进程。

范例：exit.c

6. 线程执行轨迹

1) 同步方式(非分离状态)：
   创建线程之后调用pthread_join函数等待其终止，
   并释放线程资源。

2) 异步方式(分离状态)：
   无需创建者等待，线程终止后自行释放资源。

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

使thread参数所标识的线程进入分离(DETACHED)状态。
处于分离状态的线程终止后自动释放线程资源，
且不能被pthread_join函数等待。

成功返回0，失败返回错误码。

范例：detach.c

7. 取消线程

1) 向指定线程发送取消请求

```
int pthread_cancel (pthread_t thread);
```

成功返回0，失败返回错误码。

注意：该函数只是向线程发出取消请求，
并不等待线程终止。

缺省情况下，线程在收到取消请求以后，并不会立即终止，
而是仍继续运行，直到其达到某个取消点。在取消点处，
线程检查其自身是否已被取消了，并做出相应动作。

当线程调用一些特定函数时，取消点会出现。

2) 设置调用线程的可取消状态

```
int pthread_setcancelstate (int state,
    int* oldstate);
```

成功返回0，并通过oldstate参数输出原可取消状态
(若非NULL)，失败返回错误码。

state取值：

PTHREAD_CANCEL_ENABLE - 接受取消请求(缺省)。

PTHREAD_CANCEL_DISABLE - 忽略取消请求。

3) 设置调用线程的可取消类型

```
int pthread_setcanceltype (int type, int* oldtype);
```

成功返回0，并通过oldtype参数输出原可取消类型
(若非NULL)，失败返回错误码。

type取值：

PTHREAD_CANCEL_DEFERRED   - 延迟取消(缺省)。

被取消线程在接收到取消请求之后并不立即响应，
而是一直等到执行了特定的函数(取消点)之后再响应该请求。

PTHREAD_CANCEL_ASYNCHRONOUS - 异步取消。

被取消线程可以在任意时间取消，
不是非得遇到取消点才能被取消。
但是操作系统并不能保证这一点。

范例：cancel.c

8. 线程属性

创建线程函数

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

的第二个参数即为线程属性，传空指针表示使用缺省属性。

```
typedef struct {
    // 分离状态
    //
    // PTHREAD_CREATE_DETACHED
    // - 分离线程。
    //
    // PTHREAD_CREATE_JOINABLE(缺省)
```

```
// - 可汇合线程。
//
int detachstate;

// 竞争范围
//
// PTHREAD_SCOPE_SYSTEM
// - 在系统范围内竞争资源。
//
// PTHREAD_SCOPE_PROCESS(Linux不支持)
// - 在进程范围内竞争资源。
//
int scope;

// 继承特性
//
// PTHREAD_INHERIT_SCHED(缺省)
// - 调度属性自创建者线程继承。
//
// PTHREAD_EXPLICIT_SCHED
// - 调度属性由后面两个成员确定。
//
int inheritsched;

// 调度策略
//
// SCHED_FIFO
// - 先进先出策略。
//
// 没有时间片。
//
// 一个FIFO线程会持续运行，
// 直到阻塞或有高优先级线程就绪。
//
// 当FIFO线程阻塞时，系统将其移出就绪队列，
// 待其恢复时再加到同优先级就绪队列的末尾。
//
// 当FIFO线程被高优先级线程抢占时，
// 它在就绪队列中的位置不变。
// 因此一旦高优先级线程终止或阻塞，
// 被抢占的FIFO线程将会立即继续运行。
//
// SCHED_RR
// - 轮转策略。
//
// 给每个RR线程分配一个时间片，
// 一但RR线程的时间片耗尽，
// 系统即将移到就绪队列的末尾。
//
// SCHED_OTHER(缺省)
// - 普通策略。
//
// 静态优先级为80。任何就绪的FIFO线程或RR线程，
// 都会抢占此类线程。
//
int schedpolicy;
```

```
    // 调度参数
    //
    // struct sched_param {
    //     int sched_priority; /* 静态优先级 */
    // };
    //
    struct sched_param schedparam;

    // 栈尾警戒区大小(字节)
    //
    // 缺省一页(4096字节)。
    //
    size_t guardsize;

    // 栈地址
    //
    void* stackaddr;

    // 栈大小(字节)
    //
    size_t stacksize;
}   pthread_attr_t;
```

不要手工读写该结构体，
而应调用pthread_attr_set/get函数设置/获取具体属性项。

1) 设置线程属性

第一步，初始化线程属性结构体

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

第二步，设置具体线程属性项

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

int pthread_attr_setscope (
    pthread_attr_t* attr,
    int scope);

int pthread_attr_setinheritsched (
    pthread_attr_t* attr,
    int inheritsched);

int pthread_attr_setschedpolicy (
    pthread_attr_t* attr,
    int policy);

int pthread_attr_setschedparam (
    pthread_attr_t* attr,
    const struct sched_param* param);

int pthread_attr_setguardsize (
```

```
    pthread_attr_t* attr,
    size_t guardsize);

int pthread_attr_setstackaddr (
    pthread_attr_t* attr,
    void* stackaddr);

int pthread_attr_setstacksize (
    pthread_attr_t* attr,
    size_t stacksize);

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

第三步，以设置好的线程属性结构体为参数创建线程

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

第四步，销毁线程属性结构体

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

2) 获取线程属性

第一步，获取线程属性结构体

```
int pthread_getattr_np (pthread_t thread,
    pthread_attr_t* attr);
```

第二步，获取具体线程属性项

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

int pthread_attr_getscope (
    pthread_attr_t* attr,
    int* scope);

int pthread_attr_getinheritsched (
    pthread_attr_t* attr,
    int* inheritsched);

int pthread_attr_getschedpolicy (
    pthread_attr_t* attr,
    int* policy);

int pthread_attr_getschedparam (
    pthread_attr_t* attr,
    struct sched_param* param);

int pthread_attr_getguardsize (
```

```
    pthread_attr_t* attr,
    size_t* guardsize);

int pthread_attr_getstackaddr (
    pthread_attr_t* attr,
    void** stackaddr);

int pthread_attr_getstacksize (
    pthread_attr_t* attr,
    size_t* stacksize);

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

以上所有函数成功返回0，失败返回错误码。

范例：attr.c