8 线程

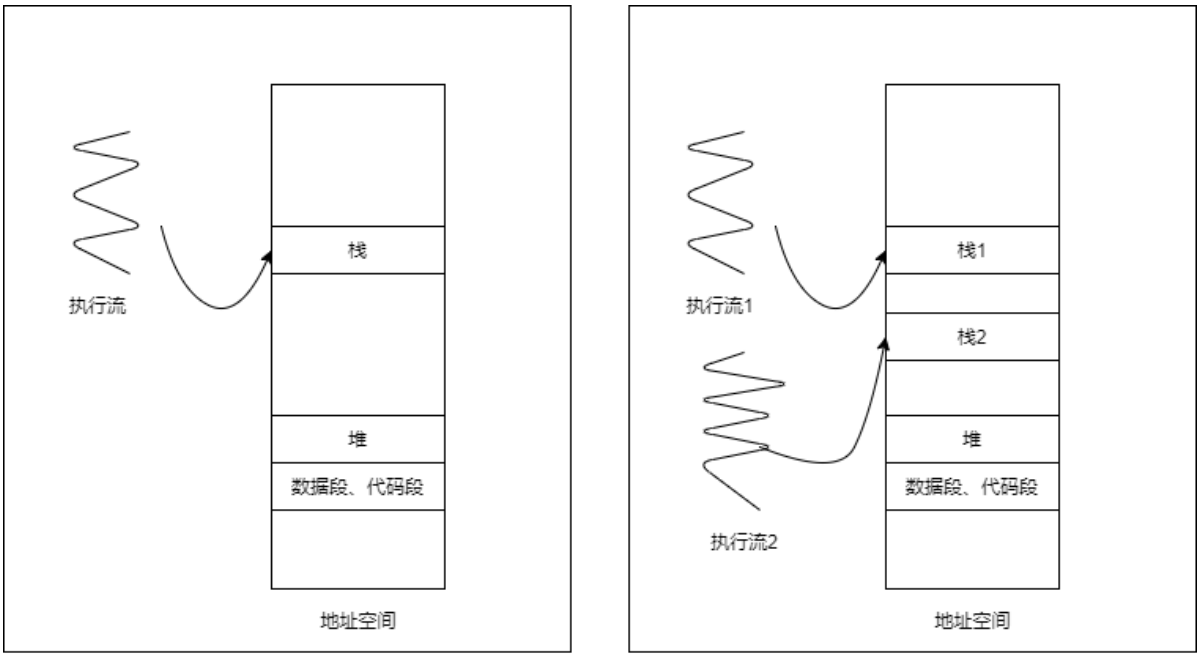
8.1 从进程到线程

在经过之前课程的学习之后，我们已经了解进程的基本概念：进程是正在执行的程序，并且是系统资源分配的基本单位。当用户需要在一台计算机上去完成多个独立的工作任务时，可以使用多进程的方式，为每个独立的工作任务分配一个进程。多进程的管理则由操作系统负责——操作系统调度进程，合理地在多个进程之间分配资源，包括CPU资源、内存、文件等等。除此以外，即便是一个单独的应用，采用多进程的设计也可以提升其**吞吐量**，改善其**响应时间**。假如在处理任务的过程中，其中一个进程因为死循环或者等待IO之类的原因无法完成任务时，操作系统可以调度另一个进程来完成任务或者响应用户的请求。比如在文本处理程序当中，可以并发地处理用户输入和保存已完成文件的任务。

随着进程进入大规模使用，程序员在分析性能的时候发现**计算机花费了大量的时间在切换不同的进程上面**。当一个进程在执行过程中，CPU的寄存器当中需要保存一些必要的信息，比如堆栈、代码段等，这些状态称作**上下文**。**上下文切换**这里涉及到大量的寄存器和内存之间的保存和载入工作。除此以外，Linux操作系统还支持虚拟内存，所以每个用户进程都拥有自己独立的地址空间，这在实现上就要求每个进程有自己独立的页目录表。所以，具体到Linux操作系统，进程切换包括两步：首先是切换页目录表，以支持一个新的地址空间；然后是进入内核态，将硬件上下文，即CPU寄存器的内容以及内核态栈切换到新的进程。

为了缩小创建和切换进程的开销，**线程**的概念便诞生了。线程又被称为**轻量级进程(Light Weight Process, LWP)**，我们将一个进程分解成多个线程，每个线程是独立的运行中程序实体，线程之间并发运行，这样线程就取代进程成为CPU时间片的分配和调度的**最小单位**，在Linux操作系统当中，每个线程都拥有自己独立的`task_struct`结构体，当然，在属于同一个进程多个线程中，`task_struct`中大量的字段是相同的或者是共享的。

注意到在Linux操作系统中，线程并没有脱离进程而存在。而计算机的内存、文件、IO设备等资源依然还是按进程为单位进行分配，属于同一个进程的多个线程会**共享进程地址空间**，每个线程在执行过程会在地址空间中有自己独立的栈，而堆、数据段、代码段、文件描述符和信号屏蔽字等资源则是共享的。



> 同一个进程的多个线程各自拥有自己的栈，这些栈虽然是独立的，但是都位于同一个地址空间中，所以一个线程可以通过地址去访问另一个线程的栈区。

因为属于同一个进程的多个线程是共享地址空间的，所以线程切换的时候不需要切换页目录表，而且上下文的内容中需要切换的部分也很少，只需要切换栈和PC指针以及其他少量控制信息即可，数据段、代码段等信息可以保持不变。因此，切换线程的花费要小于切换进程的。

> 在Linux文件系统中，路径`/proc`挂载了一个伪文件系统，通过这个伪文件系统用户可以采用访问普通文件的方式（使用read系统调用等），来访问操作系统内核的数据结构。其中，在`/proc/[num]`目录中，就包含了pid为num的进程的大量的内核数据。而在`/proc/[num]/task`就包含本进程中所有线程的内核数据。我们之前的编写进程都可以看成是单线程进程。

8.1.1 用户级线程和内核级线程

根据内核对线程的感知情况，线程可以分为用户级线程和内核级线程。操作系统内核无法调度用户级线程，所以通常会存在一个管理线程（称为运行时），管理线程负责在一个用户线程陷入阻塞的切换到另一个线程。如果线程无法陷入阻塞，则用户需要在代码当中主动调用yield以主动让出，否则一个运行中的线程将永远地占用CPU。用户级线程在CPU密集型的任务中毫无作用，而且也无法充分多核架构的优势，所以几乎所有的操作系统都支持内核级线程。

由于用户级线程在处理IO密集型任务的时候有着一定的优势，所以目前在一些服务端框架中，混合性线程也引入使用了——在这种情况下，会把用户级线程称为有栈协程。应用程序会根据硬件中CPU的核心数创建若干个内核级线程，每一个内核级线程会对应多个有栈协程。这样在触发IO操作时，不再需要陷入内核态，直接在用户态切换线程即可。

目前使用最广泛的线程库名为**NPTL** (Native POSIX Threads Library)，在Linux内核2.6版本之后，它取代了传统的LinuxThreads线程库。NPTL支持了POSIX的线程标准库，在早期，NPTL只支持用户级线程，随着内核版本的迭代，现在每一个用户级线程都对应一个内核态线程，这样NPTL线程本质就成为了**内核级线程**，可被操作系统调度。

8.2 线程的创建和终止

使用线程的思路其实和使用进程的思路类似，用户需要去关心线程的创建、退出和资源回收等行为，初学者可以参考之前学习进程的方式对比学习线程对应的库函数。下面是常用的线程库函数和之前的进程的对应关系。

线程函数	功能	类似的进程函数
pthread_create	创建一个线程	fork
pthread_exit	线程退出	exit
pthread_join	等待线程结束并回收资源	wait
pthread_self	获取线程id	getpid

在使用线程相关的函数之后，在链接时需要加上`-lpthread`选项以显式链接线程库。

8.2.1 线程函数的错误处理

之前的POSIX系统调用和库函数在调用出错的时候，通常会把全局变量`errno`设置为一个特别的数值以指示报错的类型，这样就可以调用`perror`以显示符合人阅读需求的报错信息。但是在多线程编程之中，全局变量是各个线程的共享资源，很容易被并发地读写，所以pthread系列的函数不会通过修改`errno`来指示报错的类型，它会根据不同的错误类型返回不同的返回值，使用`strerror`函数可以根据返回值显示报错字符串。

```
char *strerror(int errnum);
#define THREAD_ERROR_CHECK(ret,msg) {if(ret!=0){\
fprintf(stderr,"%s:%s\n",msg,strerror(ret));}}
```

8.2.2 创建线程

线程创建使用的函数是`pthread_create`，这个函数的函数原型如下：

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

在一个进程当中的不同线程都拥有自己独立唯一的线程id（thread id），NPTL使用pthread_t类型来保存线程id，在不同的操作系统中，pthread_t底层实现不同，具体到Linux是一个无符号整型数，使用函数`pthread_self`可以获取本线程的id。

```
//获取本线程的tid
int main()
{
    printf("pid = %d,tid = %lu\n",getpid(),pthread_self());
    return 0;
}
```

调用`pthread_create`会创建出一个新的线程。创建者需要分配空间去存储一个线程id（比如在栈帧上面），并且将其地址传入`pthread_create`函数。`attr`参数可以用来指定创建出来的线程的属性，一般可以控制其调度相关的信息，如果传入的是NULL，那么表示采用默认的线程属性。`start_routine`是一个函数指针类型的参数，其返回值和参数都是void *，`start_routine`的地位类似于主线程的main函数，它作为线程执行的入口函数。`arg`是传递给`start_routine`的参数。

```
//下面是一个不传递参数的版本
void * threadFunc(void *arg){
    printf("I am child thread, tid = %lu\n",
           pthread_self());
    return NULL;
}
int main()
{
    pthread_t tid;
    int ret = pthread_create(&tid,NULL,threadFunc,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    printf("I am main thread, tid = %lu\n",
           pthread_self());
    sleep(1);
    //usleep(20);
    //_exit(0);
    return 0;
}
```

在上面的例子当中，如果将`sleep(1)`改成`usleep(20)`，线程并发执行的特点会导致一些看上去非常奇怪的结果：在某次执行的时候，标准输出上有一定的几率显示两条相同的语句。产生这种问题的原因是这样的，stdout缓冲区在多个线程之间是共享，当执行`printf`时，会首先将stdout的内容拷贝到内核态的文件对象中，再清空缓冲区，当**主线程终止导致所有线程终止**时，可能子线程已经将数据拷贝到了内核态（此时第一份语句已经打印了），但是stdout的内容并未清空，此时进程终止，会把所有缓冲区清空，清空缓冲区的行为会将留存在缓冲区的内容直接清空并输出到标准输出中，此时就会出现内容的重复打印了。

8.2.3 线程和数据共享

多个线程是共享一片地址空间的，所以各个线程可以并发地访问的同一个数据段、堆空间和其他位置。下面是一个共享数据段的例子：

```
//多线程共享数据段
#include <func.h>
#include "test.h"
int global= 100;
void * threadFunc(void *arg){
    printf("I am child thread, tid = %lu\n",
           pthread_self());
    printf("child thread, global = %d\n", global);
    return NULL;
}
int main()
{
    pthread_t tid;
    int ret = pthread_create(&tid,NULL,threadFunc,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    printf("I am main thread, tid = %lu\n",
           pthread_self());
    global = 200;
    printf("main thread, global = %d\n", global);
    sleep(1);
    return 0;
}
```

堆空间自然也是可以共享的，我们通过会将堆空间的首地址作为参数在创建线程的时候进行传递，无论是子线程还是主线程访问的区域都是同一片空间。

```
//多线程共享堆空间
void * threadFunc(void *arg){
    char *pHeap = (char *)arg;
    printf("I am child thread, tid = %lu\n",
           pthread_self());
    strcpy(pHeap,"world");
    printf("child thread, %s\n",pHeap);
    return NULL;
}
int main()
{
    pthread_t tid;
    char *pHeap = (char *)malloc(20);
    strcpy(pHeap,"hello");
    int ret = pthread_create(&tid,NULL,threadFunc,(void *)pHeap);
```

```
    THREAD_ERROR_CHECK(ret,"pthread_create");
    printf("I am main thread, tid = %lu\n",
           pthread_self());
    sleep(1);
    printf("parent thread, %s\n",pHeap);
    free(pHeap);
    return 0;
}
```

虽然说`arg`是一个`void*`类型的参数，这暗示着用户可以使用该参数来传递一个数据的地址，但是有些情况下我们只需要传递一个整型数据，在这种情况，用户可以直接把`void*`类型的参数当成是一个8字节的普通数据(比如long)进行传递。

```
void * threadFunc(void *arg){
    printf("I am child thread, tid = %lu\n",
           pthread_self());
    //把参数当成是普通的8字节整型
    printf("child, val = %ld\n",(long)arg);
    return NULL;
}
int main()
{
    pthread_t tid;
    long val = 1001;
    int ret = pthread_create(&tid,NULL,threadFunc,(void *)val);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    val = 1002;
    ret = pthread_create(&tid,NULL,threadFunc,(void *)val);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    printf("I am main thread, tid = %lu\n",
           pthread_self());
    sleep(1);
    return 0;
}
```

需要注意的是，虽然各个线程执行的过程中拥有自己独立的栈区，但是这些所有的栈区都是在同一个地址空间当中，所以一个线程完全可以访问到另一个线程栈帧内部的数据。

```
void * threadFunc(void *arg){
    printf("I am child thread, tid = %lu\n",
           pthread_self());
    int * pval = (int *)arg;
    *pval = 1002;
    printf("child, val = %d\n",*pval);
    return NULL;
}
int main()
{
    pthread_t tid;
    int val = 1001;
    int ret = pthread_create(&tid,NULL,threadFunc,(void *)&val);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    printf("I am main thread, tid = %lu\n",
           pthread_self());
```

```
    sleep(1);
    //虽然栈帧是私有的，但是其他线程依然有权限读写
    printf("main, val = %d\n",val);
    return 0;
}
```

如果将主线程栈帧数据的地址作为参数传递给各个子线程，就一定要注意并发访问的情况，有可能另一个线程的执行会修改掉原本想要传递数据的内容。

```
//各个子线程会打印相同的结果
void * threadFunc(void *arg){
    printf("I am child thread, tid = %lu\n",
           pthread_self());
    int * pval = (int *)arg;
    printf("child, val = %d\n",*pval);
    return NULL;
}
int main()
{
    pthread_t tid;
    int val = 1001;
    int ret = pthread_create(&tid,NULL,threadFunc,(void *)&val);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    val = 1002;
    ret = pthread_create(&tid,NULL,threadFunc,(void *)&val);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    printf("I am main thread, tid = %lu\n",
           pthread_self());
    sleep(1);
    return 0;
}
```

8.2.4 线程主动退出

使用`pthread_exit`函数可以主动退出线程，无论这个函数是否是在`start_routine`中被调用，其行为类似于进程退出的`exit`。`pthread_exit`的参数是一个`void *`类型值，它描述了线程的退出状态。在`start_routine`中使用return语句可以实现类似的主动退出效果，但是其被动退出的行为有一些问题，所以使用较少。线程的退出状态可以由另一个线程使用`pthread_join`捕获，但是和进程不一样的是，另一个线程的选择是任意的，不需要是线程创建者。

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

下面是线程主动退出的例子：

```
void * threadFunc(void *arg){
    printf("I am child thread, tid = %lu\n",
           pthread_self());
    //pthread_exit(NULL);和在线程入口函数return(NULL)等价
    printf("Can you see me?\n");
}
int main()
{
    pthread_t tid;
```

```
    int ret = pthread_create(&tid,NULL,threadFunc,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    printf("I am main thread, tid = %lu\n",
           pthread_self());
    sleep(1);
    return 0;
}
```

8.2.5 获取线程退出状态

调用`pthread_join`可以使本线程处于等待状态，直到指定的`thread`终止，就结束等待，并且捕获到的线程终止状态存入`retval`指针所指向的内存空间中。因为线程的终止状态是一个`void *`类型的数据，所以`pthread_join`的调用者往往需要先申请8个字节大小的内存空间，然后将其首地址传入，在`pthread_join`的执行之后，这里面的数据会被修改。有些时候，内容可能是一片数据的首地址，还有些情况下内容就是简单的一个8字节的整型。

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

下面是例子，分别用整型数字或者20字节的堆空间来描述退出状态。

```
void * threadFunc(void *arg){
    printf("I am child thread, tid = %lu\n",
           pthread_self());
    //pthread_exit(NULL);//相当于返回成一个8字节的0
    //pthread_exit((void *)1);
    char *tret = (char *)malloc(20);
    strcpy(tret,"hello");
    return (void *)tret;
}
int main()
{
    pthread_t tid;
    int ret = pthread_create(&tid,NULL,threadFunc,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    printf("I am main thread, tid = %lu\n",
           pthread_self());
    void *tret;//在调用函数中申请void*变量
    ret = pthread_join(tid,&tret);//传入void*变量的地址
    THREAD_ERROR_CHECK(ret,"pthread_join");
    //printf("tret = %ld\n", (long) tret);
    printf("tret = %s\n", (char *)tret);
    return 0;
}
```

在使用`pthread_join`要特别注意参数的类型，`thread`参数是不需要取地址的，如果参数错误，有些情况下可能会陷入无限等待，还有些情况会立即终止，触发报错。

```
void * threadFunc(void *arg){
    printf("I am child thread, tid = %lu\n",
           pthread_self());
    pthread_exit(NULL);
}
```

```c
int main()
{
    pthread_t tid;
    int ret = pthread_create(&tid,NULL,threadFunc,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    printf("I am main thread, tid = %lu\n",
           pthread_self());
    ret = pthread_join(0,NULL);//如果第一个参数填入&tid，就会陷入无限等待
    THREAD_ERROR_CHECK(ret,"pthread_join");
    return 0;
}
```

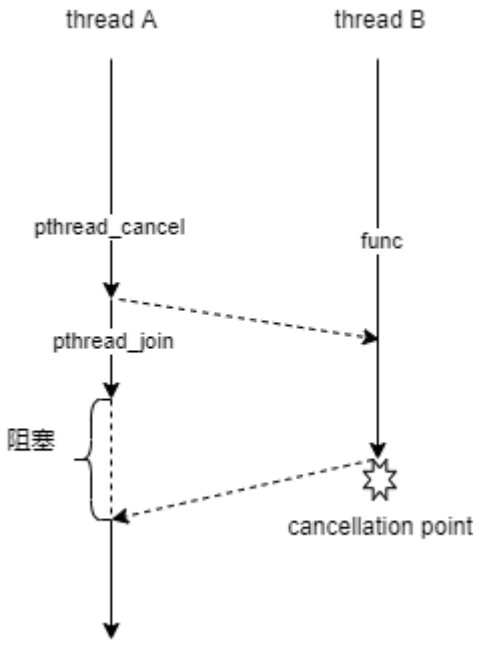
8.3 线程的取消和资源清理

8.3.1 线程的取消

线程除了可以主动退出以外，还可以被另一个线程终止。不过首先值得注意的是，我们不能轻易地在多线程程序使用信号，因为多线程是共享代码段的，从而信号处理的回调函数也是共享的，当产生一个信号到进程时，进程中用于递送信号的线程是随机的，很有可能会出现主线程因递送信号而终止从而导致所有线程异常退出的情况。

因此，要实现线程的有序退出，线程内部需要实现一种不依赖于信号的机制，这就是线程的取消`pthread_cancel`的工作职责。当一个线程调用`pthread_cancel`去取消另一个线程的时候，另一个线程会将本线程的取消标志位设置为真，当这个线程执行一些函数之时，线程就会退出。这些会导致已取消未终止的线程终止的函数称为**取消点**。

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



查看帮助手册(`pthreads(7)`)可以确定大部分取消点函数。通常来说，会引起阻塞或者是访问文件的操作就是取消点。值得注意的是，加锁`pthread_mutex_lock`和解锁函数`pthread_mutex_unlock`以及其他和锁相关的函数都不是取消点。

```
$man 7 pthreads
部分取消点：
           accept()
           close()
           connect()
           open()
           pthread_cond_timedwait()
           pthread_cond_wait()
           pthread_join()
           pthread_testcancel()
           read()
```

下面是线程取消成功的一个例子，因为子线程在受到取消信号以后，会继续运行到printf为止

```
void * threadFunc(void *arg){
    printf("I am child thread, tid = %lu\n",
           pthread_self());
    return (void *)0;
}
int main()
{
    pthread_t tid;
    int ret = pthread_create(&tid,NULL,threadFunc,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    printf("I am main thread, tid = %lu\n",
           pthread_self());
    void * pret;
    ret = pthread_cancel(tid);
    THREAD_ERROR_CHECK(ret,"pthread_cancel");
    ret = pthread_join(tid,&pret);
    THREAD_ERROR_CHECK(ret,"pthread_join");
    printf("thread return = %ld\n", (long) pret);
    return 0;
}
//可以发现pthread_cancel执行成功，因为获取的返回值是-1而不是正常退出的0
```

如果被取消的线程没有运行到取消点，那么这个线程将不会终止：

```
void * threadFunc(void *arg){
    while(1);
    return (void *)0;
}
int main()
{
    pthread_t tid;
    int ret = pthread_create(&tid,NULL,threadFunc,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    printf("I am main thread, tid = %lu\n",
           pthread_self());
    void * pret;
    ret = pthread_cancel(tid);
    THREAD_ERROR_CHECK(ret,"pthread_cancel");
    ret = pthread_join(tid,&pret);
    THREAD_ERROR_CHECK(ret,"pthread_join");
```

```
    printf("thread return = %ld\n", (long) pret);
    return 0;
}
```

使用命令`ps -elLf`可以查看线程的状态。

```
$ps -elLf
```

如果需要手打添加取消点，可以调用`pthread_testcancel`函数。

```
void pthread_testcancel(void);
```

下面是使用`pthread_testcancel`的例子

```
void *threadFunc(void *arg){
    while(1){
        pthread_testcancel();
    }
}
int main()
{
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,NULL);
    sleep(1);
    printf("sleep over!\n");
    pthread_cancel(tid);
    void *ret;
    pthread_join(tid,&ret);
    printf("You die, ret = %ld\n", (long ) ret);
    return 0;
}
```

综上，线程取消和之前的信号设计有着显著的区别，被取消的线程并不是立即退出。除此之外，线程被取消的时候还可以执行资源清理的工作。在结合资源清理行为之后，线程取消的流程大致如下：

- 首先主动方调用取消函数给被取消方发送取消信号。
- 被取消方设置取消标志为真，继续运行直到一个取消点。
- 执行线程终止清理函数，以释放占用的资源。
- 销毁thread local数据。
- 终止本线程。

8.3.2 线程资源清理

在引入线程取消之后，程序员在管理资源回收的难度上会急剧提升。为了简化资源清理行为，线程库引入了`pthread_cleanup_push`和`pthread_cleanup_pop`函数来管理线程主动或者被动终止时所申请资源（比如文件、堆空间、锁等等）。

```
void pthread_cleanup_push(void (*routine)(void *),
                          void *arg);
void pthread_cleanup_pop(int execute);
```

`pthread_cleanup_push`负责将清理函数压入一个栈中，这个栈会在下列情况下弹出：

- 线程因为取消而终止时，所有清理函数按照后进先出的顺序从栈中弹出并执行。
- 线程调用`pthread_exit`而主动终止时，所有清理函数按照后进先出的顺序从栈中弹出并执行。
- 线程调用`pthread_clean_pop`并且`execute`参数非0时，弹出栈顶的清理函数并执行。
- 线程调用`pthread_clean_pop`并且`execute`参数为0时，弹出栈顶的清理函数不执行。

值得特别注意的是：**当线程在`start_routine`中执行return语句而终止的时候，清理函数不会弹栈！**

```
void cleanup(void *p){
    free(p);
    printf("I am cleanup\n");
}
void * threadFunc(void *arg){
    void *p = malloc(100000);
    pthread_cleanup_push(cleanup,p);//一定要在cancel点之前push
    printf("I am child thread, tid = %lu\n",
           pthread_self());
    //pthread_exit((void *)0);//在pop之前exit，cleanup弹栈并被调用
    pthread_cleanup_pop(1);//在exit之后pop cleanup弹栈，如果参数为1被调用
    pthread_exit((void *)0);
}
int main()
{
    pthread_t tid;
    int ret = pthread_create(&tid,NULL,threadFunc,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    printf("I am main thread, tid = %lu\n",
           pthread_self());
    void * pret;
    //ret = pthread_cancel(tid);
    //THREAD_ERROR_CHECK(ret,"pthread_cancel");
    ret = pthread_join(tid,&pret);
    THREAD_ERROR_CHECK(ret,"pthread_join");
    printf("thread return = %ld\n", (long) pret);
    return 0;
}
```

POSIX要求`pthread_cleanup_push`和`pthread_cleanup_pop`必须成对出现在同一个作用域当中，主要是为了约束程序员在清理函数当中行为。下面是在`/urs/include/pthread.h`文件当中，线程清理函数的定义：

```
#define pthread_cleanup_push(routine, arg) \
do {                                       \
 __pthread_cleanup_class __clframe (routine, arg)

/* Remove a cleanup handler installed by the matching pthread_cleanup_push.
   If EXECUTE is non-zero, the handler function is called. */
#define pthread_cleanup_pop(execute) \
   __clframe.__setdoit (execute);                        \
} while (0)
```

可以发现push和pop的宏定义不是语义完全的，它们必须在同一作用域中成对出现才能使花括号成功匹配。

下面是压入多个清理函数的例子：

```
void cleanup(void *p){
    printf("clean up, %s\n", (char *)p);
}
void * threadFunc(void *arg){
    pthread_cleanup_push(cleanup,(void *)"first");
    pthread_cleanup_push(cleanup,(void *)"second");
    //pthread_exit((void *)0);
    return (void *)0;
    pthread_cleanup_pop(1);
    pthread_cleanup_pop(1);

}
int main()
{
    pthread_t tid;
    int ret;
    ret = pthread_create(&tid,NULL,threadFunc,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    ret = pthread_join(tid,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_join");
    return 0;
}
```

8.4 线程的同步和互斥

由于多线程之间不存在隔离，共享同一个地址在提高运行效率的同时也给用户带来了巨大的困扰。在并发执行的情况下，大量的共享资源成为竞争条件，导致程序执行的结果往往和预期的内容大相径庭，如果一个程序的结果是不正确的，那么再高的效率也毫无意义。在基于之前进程对并发的研究之上，线程库也提供了专门用于正确地访问共享资源的机制。

8.4.1 互斥锁的基本使用

在多线程编程中，用来控制共享资源的最简单有效也是最广泛使用的机制就是`mutex(MUTual EXclusion)`，即互斥锁。锁的数据类型是`pthread_mutex_t`，其本质是一个全局的标志位，线程可以对作原子地测试并修改，即所谓的**加锁**。当一个线程持有锁的时候，其余线程再尝试加锁时（包括自己再次加锁），会使自己陷入阻塞状态，直到锁被持有线程解锁才能恢复运行。所以锁在某个时刻永远不能被两个线程同时持有。

创建锁有两种形式：直接用`PHTREAD_MUTEX_INITIALIZER`初始化一个`pthread_mutex_t`类型的变量，即静态初始化锁；而使用`pthread_mutex_init`函数可以动态创建一个锁。动态创建锁的方式更加常见。

使用`pthread_mutex_destory`可以销毁一个锁。

```
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
*mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

下面是最基本的使用锁的例子，编程规范要求：**谁加锁，谁解锁。**锁的使用纷繁复杂，如果不按照规范行事，一方面会很容易出现错误，并且出错之后很难排查，另一方面，随意解锁会导致代码的可读性极差，无法后续调整优化。

```
typedef struct shareRes_s{
    pthread_mutex_t mutex;
} shareRes_t, *pShareRes_t;
void *threadFunc(void *arg){
    pthread_mutex_lock(&((pShareRes_t)arg)->mutex);
    puts("I am child thread");
    pthread_mutex_unlock(&((pShareRes_t)arg)->mutex);
}
int main()
{
    shareRes_t shared;
    pthread_t tid;
    pthread_mutex_init(&shared.mutex,NULL);
    pthread_create(&tid,NULL,threadFunc,&shared);
    sleep(1);
    //pthread_mutex_unlock(&shared.mutex);
    //虽然是可行的，但是不符合规范
    pthread_mutex_lock(&shared.mutex);
    pthread_join(tid,NULL);
    puts("I am main thread");
    return 0;
}
```

8.4.2 使用互斥锁访问共享资源

如果两个线程并发地对共享资源进行读写操作，就有可能出现竞争条件，如果在每次访问共享资源的时候都正确地使用锁（即访问之前加锁，访问之后解锁），那么程序就会得到正确的运行结果。

```
typedef struct shareRes_s{
    int val;
    pthread_mutex_t mutex;
} shareRes_t,*pshareRes_t;
void *threadFunc(void *arg){
    pshareRes_t pshared = (pshareRes_t)arg;
    for(int i = 0;i < 20000000; ++i){
        pthread_mutex_lock(&pshared->mutex);
        ++pshared->val;
        pthread_mutex_unlock(&pshared->mutex);
    }
    pthread_exit(NULL);
}
```

```
int main()
{
    int ret;
    pthread_t tid;
    shareRes_t shareRes;
    shareRes.val = 0;
    ret = pthread_mutex_init(&shareRes.mutex,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_mutex_init");
    struct timeval begTime,endTime;
    gettimeofday(&begTime,NULL);
    ret = pthread_create(&tid,NULL,threadFunc,(void *)&shareRes);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    for(int i = 0;i < 20000000; ++i){
        pthread_mutex_lock(&shareRes.mutex);
        ++shareRes.val;
        pthread_mutex_unlock(&shareRes.mutex);
    }
    ret = pthread_join(tid,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_join");
    gettimeofday(&endTime,NULL);
    long avgtime = ((endTime.tv_sec - begTime.tv_sec )* 1000000 +
(endTime.tv_usec - begTime.tv_usec));
    printf("avg lock/unlock time = %ld\n",avgtime);
    printf("val = %d\n", shareRes.val);
    ret = pthread_mutex_destroy(&shareRes.mutex);
    THREAD_ERROR_CHECK(ret,"pthread_mutex_destroy");
    return 0;
}
```

另外就是要注意到，使用线程互斥量的效率要比之前的进程间通信信号量机制好很多，所以实际工作中几乎都是使用线程互斥锁来实现互斥。

可以使用`pthread_mutex_trylock`函数来非阻塞地加锁，假如已经加锁成功，函数会直接返回。

```
typedef struct shareRes_s{
    pthread_mutex_t mutex;
} shareRes_t, *pShareRes_t;
void *threadFunc(void *arg){
    //pthread_mutex_lock(&((pShareRes_t)arg)->mutex);
    puts("I am child thread");
}
int main()
{
    shareRes_t shared;
    pthread_t tid;
    int ret;
    pthread_mutex_init(&shared.mutex,NULL);
    pthread_create(&tid,NULL,threadFunc,&shared);
    sleep(1);
    ret = pthread_mutex_trylock(&shared.mutex);
    THREAD_ERROR_CHECK(ret,"pthread_mutex_trylock");
    pthread_mutex_unlock(&shared.mutex);
    pthread_join(tid,NULL);
    puts("I am main thread");
    return 0;
```

```
}
```

8.4.3 互斥锁出错的例子

使用互斥锁的时候一定要注意访问共享资源的过程是否一定处于锁的保护下，下面是一个卖火车票的例子，它的运行有时候就会出现异常。

```
typedef struct shareRes_s{
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int ticketNum;
} shareRes_t, *pshareRes_t;
void *sellTicket1(void *arg){
    sleep(1);
    pshareRes_t pshared = (pshareRes_t)arg;
    while(pshared->ticketNum > 0){
        pthread_mutex_lock(&pshared->mutex);
        printf("Before 1 sells tickets, ticketNum = %d\n", pshared->ticketNum);
        --pshared->ticketNum;
        printf("After 1 sells tickets, ticketNum = %d\n", pshared->ticketNum);
        pthread_mutex_unlock(&pshared->mutex);
    }
    pthread_exit(NULL);
}
void *sellTicket2(void *arg){
    sleep(1);
    pshareRes_t pshared = (pshareRes_t)arg;
    while(pshared->ticketNum > 0){
        pthread_mutex_lock(&pshared->mutex);
        printf("Before 2 sells tickets, ticketNum = %d\n", pshared->ticketNum);
        --pshared->ticketNum;
        printf("After 2 sells tickets, ticketNum = %d\n", pshared->ticketNum);
        pthread_mutex_unlock(&pshared->mutex);
    }
    pthread_exit(NULL);
}

int main()
{
    shareRes_t shared;
    shared.ticketNum = 20;
    int ret;
    ret = pthread_mutex_init(&shared.mutex,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_mutex_init");
    ret = pthread_cond_init(&shared.cond,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_cond_init");
    pthread_t tid1,tid2;
    pthread_create(&tid1,NULL,sellTicket1,(void *)&shared);
    pthread_create(&tid2,NULL,sellTicket2,(void *)&shared);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    pthread_cond_destroy(&shared.cond);
    pthread_mutex_destroy(&shared.mutex);
    return 0;
```

```
}
```

上面的例子出问题的原因很简单，就是`ticketNum`这个数据并不总是在持有锁的情况下（在while循环入口的位置）被访问，这样多线程交织执行的时候就会出现竞争问题。

```
typedef struct shareRes_s{
    pthread_mutex_t mutex;
    int ticketNum;
} shareRes_t, *pShareRes_t;
void *threadFunc1(void *arg){
    pShareRes_t p = (pShareRes_t) arg;
    int mySell = 0;
    while(1){
        pthread_mutex_lock(&p->mutex);
        if(p->ticketNum <= 0){
            pthread_mutex_unlock(&p->mutex);
            printf("I am 1, sell = %d\n", mySell);
            break;
        }
        --p->ticketNum;
        ++mySell;
        pthread_mutex_unlock(&p->mutex);
        //sleep(1);
    }
    pthread_exit(NULL);
}
void *threadFunc2(void *arg){
    pShareRes_t p = (pShareRes_t) arg;
    int mySell = 0;
    while(1){
        pthread_mutex_lock(&p->mutex);
        if(p->ticketNum <= 0){
            pthread_mutex_unlock(&p->mutex);
            printf("I am 2, sell = %d\n", mySell);
            break;
        }
        --p->ticketNum;
        ++mySell;
        pthread_mutex_unlock(&p->mutex);
        //sleep(1);
    }
    pthread_exit(NULL);
}
int main()
{
    shareRes_t shared;
    pthread_mutex_init(&shared.mutex,NULL);
    //shared.ticketNum = 2000;
    shared.ticketNum = 20000000;
    pthread_t tid1,tid2;
    pthread_create(&tid1,NULL,threadFunc1,&shared);
    pthread_create(&tid2,NULL,threadFunc2,&shared);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    puts("I am main thread");
```

```
    return 0;
}
```

8.4.4 死锁

使用互斥锁的时候必须小心谨慎，如果是需要持续多个锁的情况，加锁和解锁之间必须要维持一定的顺序。即使是只有一把锁，如果使用不当，也会导致死锁。当一个线程持有了锁以后，又试图对同一把锁加锁时，线程会陷入死锁状态。

可以使用`pthread_mutexattr_settype`函数修改锁的属性，检错锁在重复加锁是会报错，递归锁或者称为可重入锁在重复加锁不会死锁时，只是会增加锁的引用计数，解锁时也只是减少锁的引用计数。但是在实际工作中，如果一个设计必须依赖于递归锁，那么这个设计肯定是有问题的。

```
typedef struct shareRes_s{
    pthread_mutex_t mutex;
} shareRes_t, *pShareRes_t;
void *threadFunc(void *arg){
    pShareRes_t p = (pShareRes_t)arg;
    pthread_mutex_lock(&p->mutex);
    puts("fifth");
    pthread_mutex_unlock(&p->mutex);
}
int main()
{
    shareRes_t shared;
    int ret;
    pthread_mutexattr_t mutexattr;
    pthread_mutexattr_init(&mutexattr);
    //ret = pthread_mutexattr_settype(&mutexattr,PTHREAD_MUTEX_ERRORCHECK);
    ret = pthread_mutexattr_settype(&mutexattr,PTHREAD_MUTEX_RECURSIVE);
    THREAD_ERROR_CHECK(ret,"pthread_mutexattr_settype");
    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,(void *)&shared);
    pthread_mutex_init(&shared.mutex,&mutexattr);
    ret = pthread_mutex_lock(&shared.mutex);
    THREAD_ERROR_CHECK(ret,"pthread_mute_lock 1");
    puts("first");
    ret = pthread_mutex_lock(&shared.mutex);
    THREAD_ERROR_CHECK(ret,"pthread_mute_lock 2");
    puts("second");
    pthread_mutex_unlock(&shared.mutex);
    puts("third");
    pthread_mutex_unlock(&shared.mutex);//两次加锁，要有两次解锁
    puts("forth");
    pthread_join(tid,NULL);
    return 0;
}
```

> 我们目前所使用的mutex互斥锁是一种睡眠锁，不满足条件的线程会陷入睡眠状态。还存在另一个锁称作自旋锁，当线程不满足条件时，线程会一直死循环直到条件成立。一般适合用于条件很快就会被满足的情况。实际上操作系统内核实现互斥锁的底层就用了自旋锁。

8.4.5 同步和条件变量

理论上来说，利用互斥锁可以解决所有的同步问题，但是生产实践之中往往会出现这样的问题：一个线程能否执行取决于业务的状态，而该状态是多线程共享的，状态的数值会随着程序的运行不断地变化，线程也经常在可运行和不可运行之间动态切换。假如单纯使用互斥锁来解决问题的话，就会出现大量的重复的"加锁-检查条件不满足-解锁"的行为，这样的话，不满足条件的线程会经常试图占用CPU资源，上下文切换也会非常频繁。

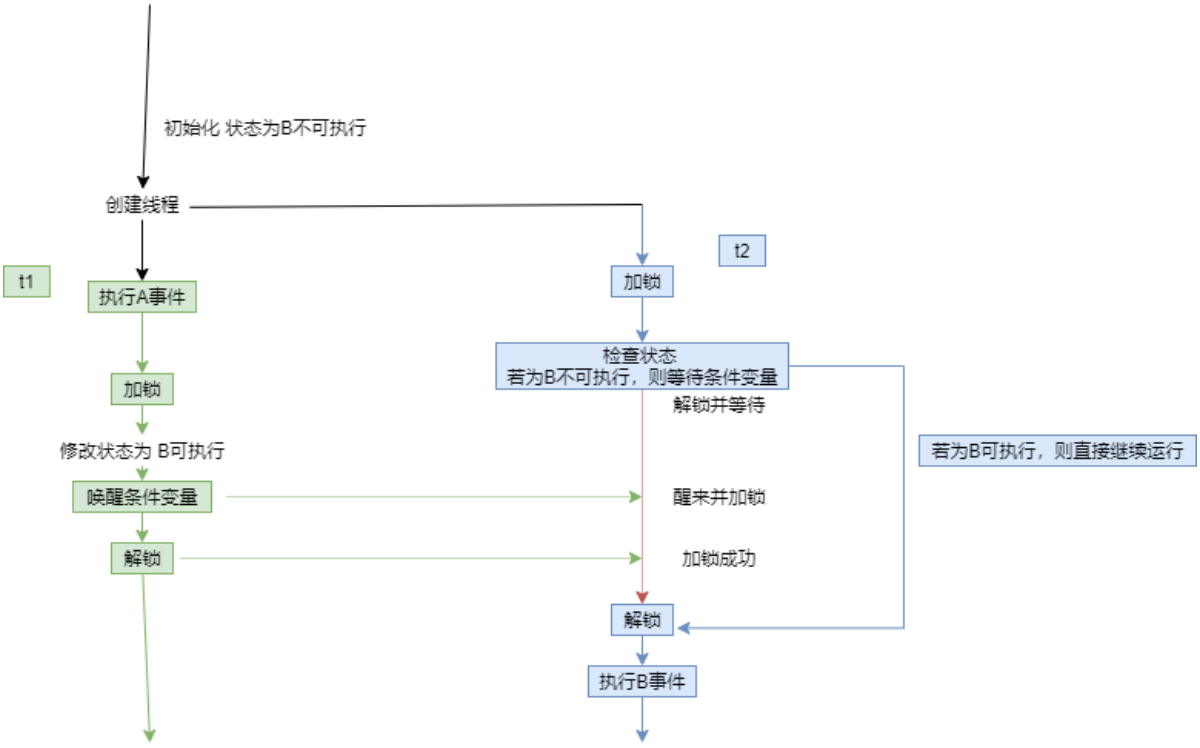
对于这样依赖于共享资源这种条件来控制线程之间的同步的问题，我们希望采用一种**无竞争**的方式让多个线程在共享资源处会和——这就是**条件变量**。当涉及到同步问题时，业务上需要设计一个**状态/条件**（一般是一个标志位）来表示该线程到底是可运行还是不可运行的，这个状态是多线程共享的，故修改的时候必须加锁访问，这就意味着**条件变量一定要配合锁来使用**。条件变量是一种"等待-唤醒"机制：线程运行的时候发现不满足执行的状态可以**等待**，线程运行的时候如果修改了状态可以**唤醒**其他线程。

接下来我们来举一个条件变量的例子。

业务场景：假设有两个线程t1和t2并发运行，t1会执行A事件，t2会执行B事件，现在业务要求，无论t1或t2哪个线程先占用CPU，总是需要满足A先B后的同步顺序。

解决方案：

- 在t1和t2线程执行之前，先**初始化**状态为**B不可执行**。
- t1线程执行A事件，执行完成以后先加锁，**修改状态为B可执行**，并唤醒条件变量，然后解锁（解锁和唤醒这两个操作可以交换顺序）；
- t2线程先加锁，然后检查状态：假如B不可执行，则B阻塞在条件变量上，当t2阻塞在条件变量以后，t2线程会解锁并陷入阻塞状态直到t1线程唤醒为止，t2被唤醒以后，会先加锁。接下来t2线程处于加锁状态，可以在解锁之后，再来执行B事件；而假如状态为B可执行，则t2线程处于加锁状态继续执行后续代码，也可以在解锁之后，再来执行B事件。



通过上面的设计，可以保证无论t1和t2按照什么样的顺序交织执行，A事件总是先完成，B事件总是后完成——即使是比较极端的情况也是如此：比如t1一直占用CPU直到结束，那么t2占用CPU时，状态一定是B可执行，则不会陷入等待；又比如t2先一直占用CPU，t2检查状态时会发现状态为B不可执行，就会阻塞在条件变量之上，这样就要等到A执行完成以后，才能醒来继续运行。

下面是具体条件变量相关接口：

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;//静态初始化
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);//动态初始化
int pthread_cond_signal(pthread_cond_t *cond); //唤醒一个线程
int pthread_cond_broadcast(pthread_cond_t *cond);//唤醒所有线程
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);//等待
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);//具有超时时限的等待
int pthread_cond_destroy(pthread_cond_t *cond);//条件变量销毁
```

- `pthread_cond_t`是条件变量的数据类型，可以采用静态方式初始化，也可以使用`pthread_cond_init`进行动态初始化。
- `pthread_cond_wait`用于使本线程陷入阻塞，前提是自己必须处于加锁状态，`pthread_cond_timedwait`效果类似，只不过可以指定一个最大超时时间。
- `pthread_cond_signal`和`pthread_cond_broadcast`分别采用单播和广播的形式唤醒阻塞在条件变量上的线程，值得注意的是这两个函数是非阻塞的，所以如果调用时没有线程阻塞的话，将不会产生任何效果。
- `pthread_cond_destroy`可以用于销毁条件变量。

以下是刚才示例的实现代码：

```
typedef  struct shareRes_s {
    int flag; // flag 0 A可以执行 1 B可以执行
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} shareRes_t;
void A(){
    printf("Before A\n");
    sleep(3);
    printf("After A\n");
}
void B(){
    printf("Before B\n");
    sleep(3);
    printf("After B\n");
}
void *threadFunc(void *arg){
    sleep(5);
    shareRes_t * pshareRes = (shareRes_t *)arg;
    pthread_mutex_lock(&pshareRes->mutex);
    if(pshareRes->flag != 1){
        pthread_cond_wait(&pshareRes->cond,&pshareRes->mutex);
    }
    pthread_mutex_unlock(&pshareRes->mutex);
    B();
    pthread_exit(NULL);
```

```
}
int main(int argc, char *argv[])
{
    shareRes_t shareRes;
    shareRes.flag = 0;
    pthread_mutex_init(&shareRes.mutex, NULL);
    pthread_cond_init(&shareRes.cond, NULL);

    pthread_t tid;
    pthread_create(&tid,NULL,threadFunc,&shareRes);

    A();
    pthread_mutex_lock(&shareRes.mutex);
    shareRes.flag = 1;
    pthread_cond_signal(&shareRes.cond);
    pthread_mutex_unlock(&shareRes.mutex);

    pthread_join(tid,NULL);
    return 0;
}
```

下面是一个复杂的卖火车票的例子，我们除了拥有两个卖票窗口之外，还会有一个放票部门。

```
typedef struct shareRes_s{
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int ticketNum;
} shareRes_t, *pshareRes_t;
void *sellTicket1(void *arg){
    sleep(1);//让setTicket先抢到锁
    pshareRes_t pshared = (pshareRes_t)arg;
    while(1){
        pthread_mutex_lock(&pshared->mutex);
        if(pshared->ticketNum > 0){
            printf("Before 1 sells tickets, ticketNum = %d\n", pshared-
>ticketNum);
            --pshared->ticketNum;
            if(pshared->ticketNum == 0){
                pthread_cond_signal(&pshared->cond);
            }
            usleep(500000);
            printf("After 1 sells tickets, ticketNum = %d\n", pshared-
>ticketNum);
            pthread_mutex_unlock(&pshared->mutex);
            usleep(200000);//等待一会，让setTicket抢到锁
        }
        else{
            pthread_mutex_unlock(&pshared->mutex);
            break;
        }
    }
    pthread_exit(NULL);
}
void *sellTicket2(void *arg){
    sleep(1);
```

```
    pshareRes_t pshared = (pshareRes_t)arg;
    while(1){
        pthread_mutex_lock(&pshared->mutex);
        if(pshared->ticketNum > 0){
            printf("Before 2 sells tickets, ticketNum = %d\n", pshared-
>ticketNum);
            --pshared->ticketNum;
            if(pshared->ticketNum == 0){
                pthread_cond_signal(&pshared->cond);
            }
            usleep(500000);
            printf("After 2 sells tickets, ticketNum = %d\n", pshared-
>ticketNum);
            pthread_mutex_unlock(&pshared->mutex);
            usleep(200000);
        }
        else{
            pthread_mutex_unlock(&pshared->mutex);
            break;
        }
    }
    pthread_exit(NULL);
}
void *setTicket(void *arg){
    pshareRes_t pshared = (pshareRes_t)arg;
    pthread_mutex_lock(&pshared->mutex);
    if(pshared->ticketNum > 0){
        printf("Set is waiting\n");
        int ret = pthread_cond_wait(&pshared->cond,&pshared->mutex);
        THREAD_ERROR_CHECK(ret,"pthread_cond_wait");
    }
    printf("add tickets\n");
    pshared->ticketNum = 10;
    pthread_mutex_unlock(&pshared->mutex);
    pthread_exit(NULL);
}
int main()
{
    shareRes_t shared;
    shared.ticketNum = 20;
    int ret;
    ret = pthread_mutex_init(&shared.mutex,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_mutex_init");
    ret = pthread_cond_init(&shared.cond,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_cond_init");
    pthread_t tid1,tid2,tid3;
    pthread_create(&tid3,NULL,setTicket,(void *)&shared);//希望setTicket第一个执行
    pthread_create(&tid1,NULL,sellTicket1,(void *)&shared);
    pthread_create(&tid2,NULL,sellTicket2,(void *)&shared);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    pthread_join(tid3,NULL);
    pthread_cond_destroy(&shared.cond);
    pthread_mutex_destroy(&shared.mutex);
    return 0;
}
```

8.5 线程的属性

在线程创建的时候，用户可以给线程指定一些属性，用来控制线程的调度情况、CPU绑定情况、屏障、线程调用栈和线程分离等属性。这些属性可以通过一个`pthread_attr_t`类型的变量来控制，可以使用`pthread_attr_set`系列设置属性，然后可以传入`pthread_create`函数，从控制新建线程的属性。

在这里，我们以`pthread_attr_setdetachstate`为例子演示如何设置线程的属性。

```
void * threadFunc(void *arg){
    return NULL;
}
int main()
{
    pthread_t tid;
    pthread_attr_t tattr;
    pthread_attr_init(&tattr);
    pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
    int ret = pthread_create(&tid,&tattr,threadFunc,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    ret = pthread_join(tid,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_join");
    pthread_attr_destroy(&tattr);
    return 0;
}
```

分离属性影响一个线程的终止状态是否能被其他线程使用`pthread_join`函数捕获终止状态。如果一个线程设置了分离属性，那么另一个线程使用`pthread_join`时会返回一个报错。

8.6 线程安全与可重入性

8.6.1 线程安全

由于多线程之间是共享同一个进程地址空间，所以多线程在访问共享数据的时候会出现竞争问题，这个问题不只会发生在用户自定义函数中，在一些库函数执行中也可能会出现竞争问题。有些库函数在设计的时候会申请额外的内存，或者会在静态区域分配数据结构——一个典型的库函数就是`ctime`。`ctime`函数会把日历时间字符串存储在静态区域。

```
void * threadFunc(void *arg){
    time_t now;
    time(&now);
    char *p = ctime(&now);
    printf("child ptime = %s\n",p);
    sleep(5);
    printf("child ptime = %s\n",p);
    pthread_exit(NULL);
}
int main()
{
    pthread_t tid;
    int ret = pthread_create(&tid,NULL,threadFunc,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_create");
```

```
    sleep(2);
    time_t mainNow;
    time(&mainNow);
    printf("main time = %s\n",ctime(&mainNow));
    ret = pthread_join(tid,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_join");
    return 0;
}
```

在上述例子中的，`p`指向的区域是静态的，所以即使子线程没有作任何修改，但是因为主线程会调用`ctime`修改静态区域的字符串，子线程两次输出的结构会有不同。使用`ctime_r`可以避免这个问题，`ctime_r`函数会增加一个额外指针参数，这个指针可以指向一个线程私有的数据，比如函数栈帧内，从而避免发生竞争问题。

```
void *threadFunc(void *arg){
    time_t now;
    time(&now);
    char buf[256];
    char *p = ctime_r(&now,buf);
    printf("child ptime = %s\n",p);
    sleep(5);
    printf("child ptime = %s\n",p);
    pthread_exit(NULL);
}
int main()
{
    pthread_t tid;
    int ret = pthread_create(&tid,NULL,threadFunc,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_create");
    sleep(2);
    char buf[256];
    time_t mainNow;
    time(&mainNow);
    printf("main time = %s\n",ctime_r(&mainNow,buf));
    ret = pthread_join(tid,NULL);
    THREAD_ERROR_CHECK(ret,"pthread_join");
    return 0;
}
```

类似于`ctime_r`这种函数是**线程安全**的，如果额外数据是分配在线程私有区域的情况下，在多线程的情况下并发地使用这些库函数是不会出现并发问题的。在帮助手册中，库函数作者会说明线程的安全属性。

8.6.2 可重入性

在信号、多线程这些情况下，一个函数在执行过程中，有可能会异步地再重新调用同一个函数。如果重复的函数调用有可能会导致错乱的结果，那么这些函数就是**不可重入**的。可重入函数和线程安全函数几乎是同义词的。下面是一个使用信号的情况下，可重入函数和不可重入函数的例子。

```
char* funcA(int num)
{
    static char p[20]={0} ;
    sprintf(p,"%d",num);
```

```
    sleep(3);
    return p;
}
//B函数可重入
char* funcB(int num,char* p)
{
    sprintf(p,"%d",num);
    sleep(3);
    return p;
}
void sigFunc(int num)
{
    printf("signum=%d\n",num);
    funcA(2);
    char buf[64]={0};
    funcB(3,buf);
}
int main()
{
    signal(2,sigFunc);
    char* p;
    p = funcA(1);
    printf("p=%s\n",p);
    char buf[64]={0};
    p = funcB(1,buf);
    printf("p=%s\n",p);
    return 0;
}
```

理论上来说，实现可重入函数有下列需求：

- 访问静态或者全局数据必须采取同步手段。
- 不能调用非可重入函数。

一个比较典型的不可重入函数例子就是`malloc`函数，`malloc`函数必然是要修改静态数据的，为了保证线程安全性，`malloc`函数的实现当中会存在加锁和解锁的过程，假如`malloc`执行到加锁之后，解锁之前的时候，此时有信号产生并且递送的话，线程会转向执行信号处理回调函数，假如信号处理函数当中又调用了`malloc`函数，此时就会导致死锁——这就是`malloc`的不可重入性。