## 线程使用场景

(1)流水线方式。根据业务特点，将一个流程的处理分割成多个线程，形成流水线的处理方式。产生的结果:延长单一流程的处理时间，提高系统整体的吞吐能力。  
(2)线程池方式。针对处理时间比较长且没有内蕴状态的线程，使用线程池方式分流消息，加快对线程消息的处理，避免其成为系统瓶颈。  
线程使用的关键是线程消息队列、线程锁、智能指针的使用。其中以线程消息队列最为重要。

## 线程消息队列描述

线程消息队列，就是一个普通的循环队列加上“多生产者－单（多）消费者的存/取操作”。流水线方式中的线程是单消费者，线程池方式中的线程是多消费者。

为了后文更好的描述问题，作如下说明：

(1) 假定循环队列queue中, 入队操作put\_queue, 出队操作get\_queue。

(2) 生产者消费者：生产者线程生产消息，放在一个空缓冲区中，供消费者线程消费，生产者生产消息(put\_queue)，如果缓冲区满，则被阻塞，消费者消费消息(get\_queue)，如果缓冲区空，则被阻塞。线程消息队列就是生产者消费者问题中的缓冲区，而它的生产者是不限定的，任何线程都可以作为生产者向其中进行put\_queue操作，消费线程则可能是一个，也可能是多个。因此对循环队列的任何操作都要加锁，以保证线程安全。

## 线程相关的操作

1. pthread\_t类型的创建、属性创建设置等。

这类具体可以: man pthread\_creat; man pthread\_attr\_init; man pthread\_detach; man pthread\_join等查看

1. pthread\_mutex\_t类型的操作。

这类具体可以: man pthread\_mutex\_init可以看到所有相关的操作。

1. pthread\_cond\_t类型的操作。man pthread\_cond\_init。pthread\_cond\_t的wait和signal操作一定要和pthread\_mutex\_t的lock、unlock配合使用。类似于此：

## linux的线程库

2.6之后的内核的默认使用的是redhat公司的NPTL（原生posix线程库），以前内核使用的是LinuxThreads库，两者的简单介绍可以看<http://www.ibm.com/developerworks/cn/linux/l-threading.html>。不过对于应用者，分析两者的区别和优劣也没什么大意义。这里特别提下NPTL的futex机制。借助该机制，pthread\_mutex的性能大大提高，只要不进入竞争态，进程就不会陷入内核态。这点可以自己写示例程序，通过strace -c 跟踪进程的系统调用，另外还可以证实总是进入内核态的操作有pthread\_cond\_signal和sem\_post。

## 通过上面的分析，我们可以有如下结论：

1. 减少pthread\_cond\_signal和sem\_post的调用，只在有必要的时候调用；
2. 尽量避免pthread\_mutex进入竞争态。增大消息队列的大小，可以有效减少竞态条件的出现。

## 实用的线程消息队列实现(msg\_queue.h)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <pthread.h>

pthread\_mutex\_t mux;

pthread\_cond\_t cond\_get, cond\_put;

struct msg\_queue {

void\*\* buffer; // 缓冲数据, .buffer = msg

int size; // 队列大小，使用的时候给出稍大的size，可以减少进入内核态的操作

int lget; // 取队列数据的偏移量

int lput; // 放队列数据的偏移量

int nData; // 队列中数据的个数,用来判断队列满/空

int nFullThread; // 由于队列满而阻塞在put\_queue的线程个数

int nEmptyThread; // 由于队列空而阻塞在get\_queue的线程个数

};

void\* get\_queue(struct msg\_queue \*q){

void\* data = NULL;

pthread\_mutex\_lock(&mux);

while(q->lget == q->lput && 0 == q->nData){

// 此处循环判断的原因是：假设2个消费者线程在get\_queue阻塞，然后两者都被激活，

// 而其中一个线程运行比较块，快速消耗了2个数据，另一个线程醒来的时候已

// 经没有新数据可以消耗了。这种情况是有可能的：比如，其它生产者线程快速

// 调用put\_queue两次，如果有2个线程在get\_queue处阻塞，就会被同时激活，

// 而完全有可能，其中一个被激活的线程获取到了cpu，快速处理了2个消息。

// 对于循环队列，如果lget与lput相等，那么只有两种情况，

// 1：nData不为0，队列满

// 2：nData为0，队列空

q->nEmptyThread++;

pthread\_cond\_wait(&cond\_get, &mux);

q->nEmptyThread--;

}

#ifdef DEBUG

printf("get data! lget:%d", q->lget);

#endif

data = (q->buffer)[q->lget++];

if(q->lget == q->size){

// queue用作循环队列

q->lget = 0;

}

q->nData--;

#ifdef DEBUG

printf(" nData:%d\n", q->nData);

#endif

if(q->nFullThread){

// 仅在必要时才调用pthread\_cond\_signal, 尽量少陷入内核态

pthread\_cond\_signal(&cond\_put);

}

pthread\_mutex\_unlock(&mux);

return data;

}

void put\_queue(struct msg\_queue \*q, void\* data){

pthread\_mutex\_lock(&mux);

while(q->lget == q->lput && q->nData){

q->nFullThread++;

pthread\_cond\_wait(&cond\_put, &mux);

q->nFullThread--;

}

#ifdef DEBUG

printf("put data! lput:%d", q->lput);

#endif

(q->buffer)[q->lput++] = data;

if(q->lput == q->size){

q->lput = 0;

}

q->nData++;

#ifdef DEBUG

printf(" nData:%d\n", q->nData);

#endif

if(q->nEmptyThread){

pthread\_cond\_signal(&cond\_get);

}

pthread\_mutex\_unlock(&mux);

}