组会报告

徐益

2018/4/26

1 本周学习内容

1. 《多核应用编程实战》1-5 章
2. LTE 数据处理多核版本阅读源码

2 《多核应用编程实战》

2.1 多线程的特点

2.1.1 时序特点

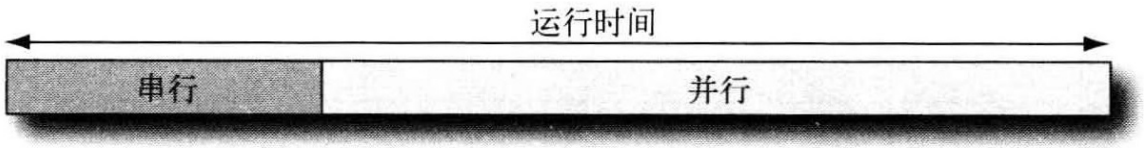

图 1: 单线程时序特点

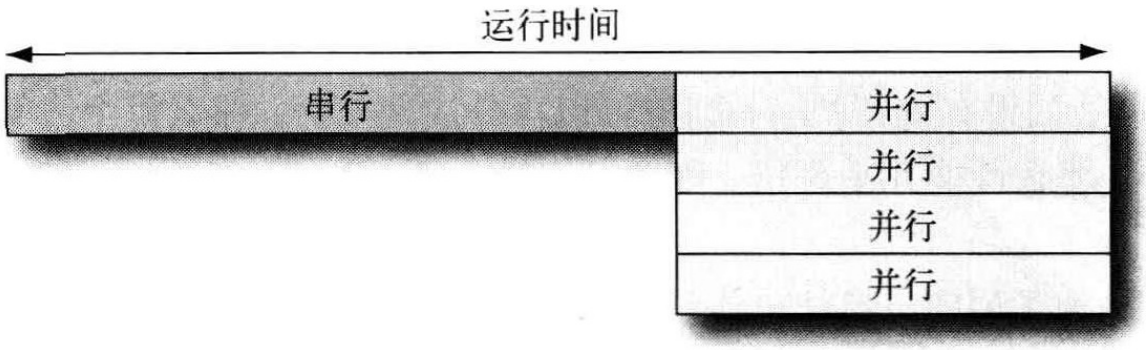

图 2: 多线程时序特点

2.1.2 内存空间上的特点

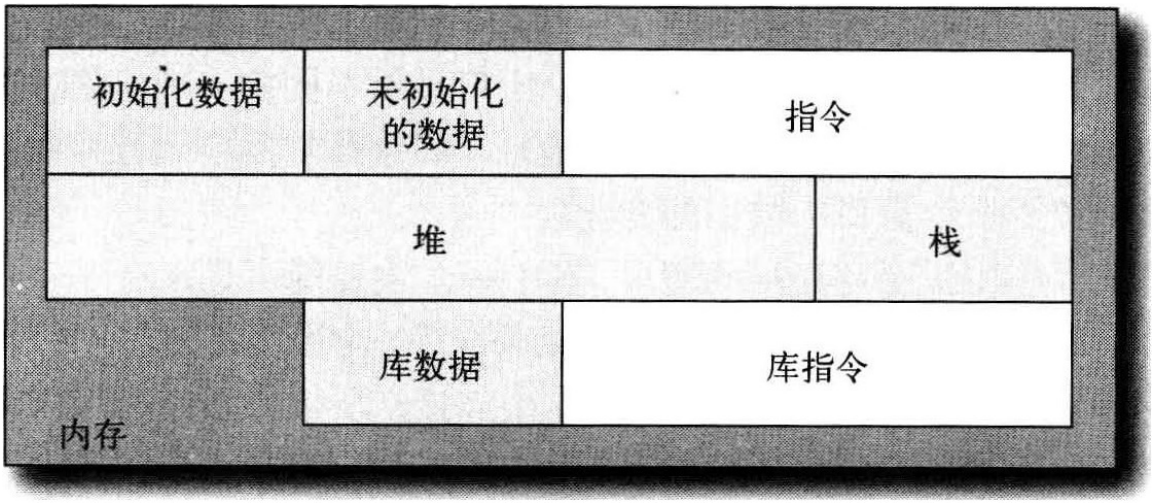

图 3: 单线程内存空间特点

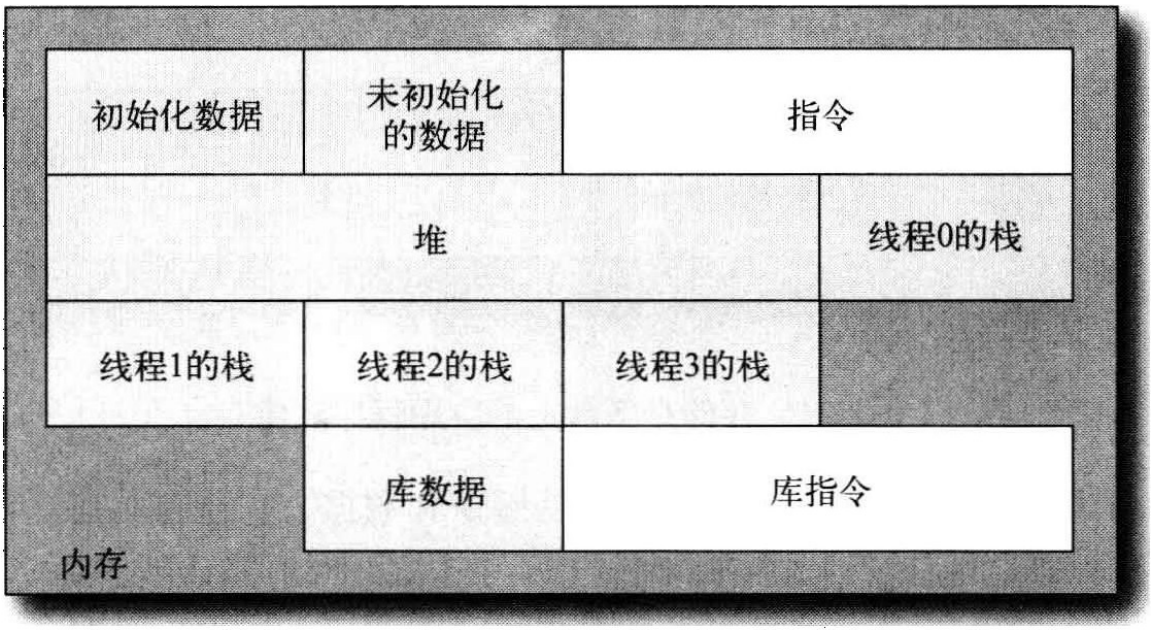

图 4: 多线程内存空间特点

2.2 同步与数据共享相关原语

原语	含义
互斥量	相互排斥的锁，一次只有一个线程获得互斥锁，从而保证对变量的访问
临界区	获取和释放互斥锁之间的代码区域
自旋锁	本质上是互斥锁，等待获取自旋锁的线程会不断尝试，等待获取互斥锁的线程会进入休眠
信号量	可递增或递减的计数器，用于对资源存在有限限制的情况
读写锁	解决修改共享数据时的数据争用问题
屏障	令各线程等待直到所有线程完成
条件变量	线程间用于沟通准备情况的变量，条件为真时唤醒线程

2.3 POSIX 线程函数

2.3.1 基本函数

函数	功能
pthread_create()	创建新线程
pthread_exit()	主线程等待所有子线程终止后退出
pthread_join()	待线程结束，回收线程，获得返回值
pthread_detch()	使线程分离（无需等待主线程回收返回值）
pthread_attr_init()	初始化 pthread 属性结构
pthread_attr_destroy()	销毁 pthread 属性结构

2.3.2 原语相关函数

函数	功能
pthread_mutex_init()	互斥锁初始化
pthread_mutex_destroy()	互斥锁销毁
pthread_mutex_lock()	获取互斥锁
pthread_mutex_unlock()	释放互斥锁
pthread_spin_init()	自旋锁初始化
pthread_spin_destroy()	自旋锁销毁
pthread_spin_lock()	获取自旋锁
pthread_spin_unlock()	释放自旋锁
pthread_rwlock_init()	读写锁初始化
pthread_rwlock_destroy()	读写锁销毁
pthread_rwlock_rdlock()	获取读锁
pthread_rwlock_rdunlock()	释放读锁
pthread_rwlock_wrlock()	获取写锁
pthread_rwlock_wrunlock()	释放写锁
pthread_barrier_init()	屏障初始化
pthread_barrier_destroy()	屏障销毁
pthread_barrier_wait()	在到达屏障的线程达到一定数量后返回
sem_init()	信号量初始化
sem_destroy()	信号量销毁
sem_open()	打开命名信号量
sem_close()	关闭命名信号量
sem_unlink()	删除命名信号量
sem_wait()	尝试减少非命名信号量，信号量为 0 时等待，直到非零时返回
sem_post()	增加非命名信号量
pthread_cond_init()	条件变量初始化
pthread_cond_destroy()	条件变量销毁
pthread_cond_signal()	产生唤醒信号
pthread_cond_broadcast()	广播唤醒信号
pthread_cond_wait()	等待唤醒信号

3 LTE 数据处理多线程版本源码

3.1 thread_pool.h

3.1.1 void pool_init(int coreId_start, int _threadNum, int pool_index);

功能：初始化线程池

参数：

参数	含义
int coreId_start	线程初始序号
int threadNum	线程池中线程数目
int pool_index	线程池序号

3.1.2 void pool_add_task(Fun myfun, void *arg, int pool_index);

功能：在线程池中添加任务

参数：

参数	含义
Fun myfun	函数名
void *arg	函数参数
int pool_index	线程池序号

3.1.3 void pool_destroy(int pool_index);

功能：待所有线程完成后销毁线程及线程池

参数：

参数	含义
int pool_index	线程池序号

3.2 main.c

```
1  const int threadNum_tx = 1;                  // 发送端线程数
2  const int threadNum_rx = 1;                  // 接收端线程数
3
4  /* 声明互斥锁 */
5  pthread_mutex_t mutex1_tx;                   // crc_cbsegm中的互斥锁
6  pthread_mutex_t mutex2_tx;                   // crc_mod中的互斥锁
7  pthread_mutex_t mutex1_rx;                   // chest_calsym中的互斥锁
8  pthread_mutex_t mutex2_rx;                   // derm_crc中的互斥锁
9  pthread_mutex_t mutex3_rx;                   // crc_check中的互斥锁
10
11 /* 声明信号量 */
12 sem_t exit_signal;                           // 收发端主线程完成数信号量
13 //test
14 sem_t tx_signal;                            // 开启发送端主线程的信号量
15 sem_t rx_signal;                            // 开启接收端主线程的信号量
16
17 int main(){
18
19         /* 初始化互斥锁 */
20         pthread_mutex_init(&mutex1_tx,NULL);
```

```
21          pthread_mutex_init(&mutex2_tx,NULL);
22          pthread_mutex_init(&mutex1_rx,NULL);
23          pthread_mutex_init(&mutex2_rx,NULL);
24          pthread_mutex_init(&mutex3_rx,NULL);
25
26          /* 初始化信号量 */
27          sem_init(&exit_signal,0,0);
28          sem_init(&tx_signal,0,0);
29          sem_init(&rx_signal,0,0);
30
31          /* 初始化线程池 */
32          pool_init(0, 1, 0);                                 // 发送端主线程池
33          printf("creat␣pool␣0...\n");
34          pool_init(1, 1, 1);                                 // 接收端主线程池
35          printf("creat␣pool␣1...\n");
36          pool_init(2, threadNum_tx, 2);                      // 发送端任务线程池
37          printf("creat␣pool␣2...\n");
38          pool_init(2 + threadNum_tx, threadNum_rx, 3);       // 接收端任务线程池
39          printf("creat␣pool␣3...\n");
40
41          /* 添加发送端主任务 */
42          pool_add_task(TaskScheduler_tx, NULL, 0);
43          printf("add␣tx␣taskScheduler␣to␣pool␣0...\n");
44          /* 添加接收端主任务 */
45          pool_add_task(TaskScheduler_rx, NULL, 1);
46          printf("add␣rx␣taskScheduler␣to␣pool␣1...\n");
47
48          /* 等待收发机主任务完成 */
49          for(int i = 0; i < 2; i++) sem_wait(&exit_signal);
50
51          /* 销毁线程池 */
52          pool_destroy(0);
53          pool_destroy(1);
54
55          /* 销毁信号量*/
56          sem_destroy(&exit_signal);
57
58          return 0;
59  }
```

3.3 收发主线程流程

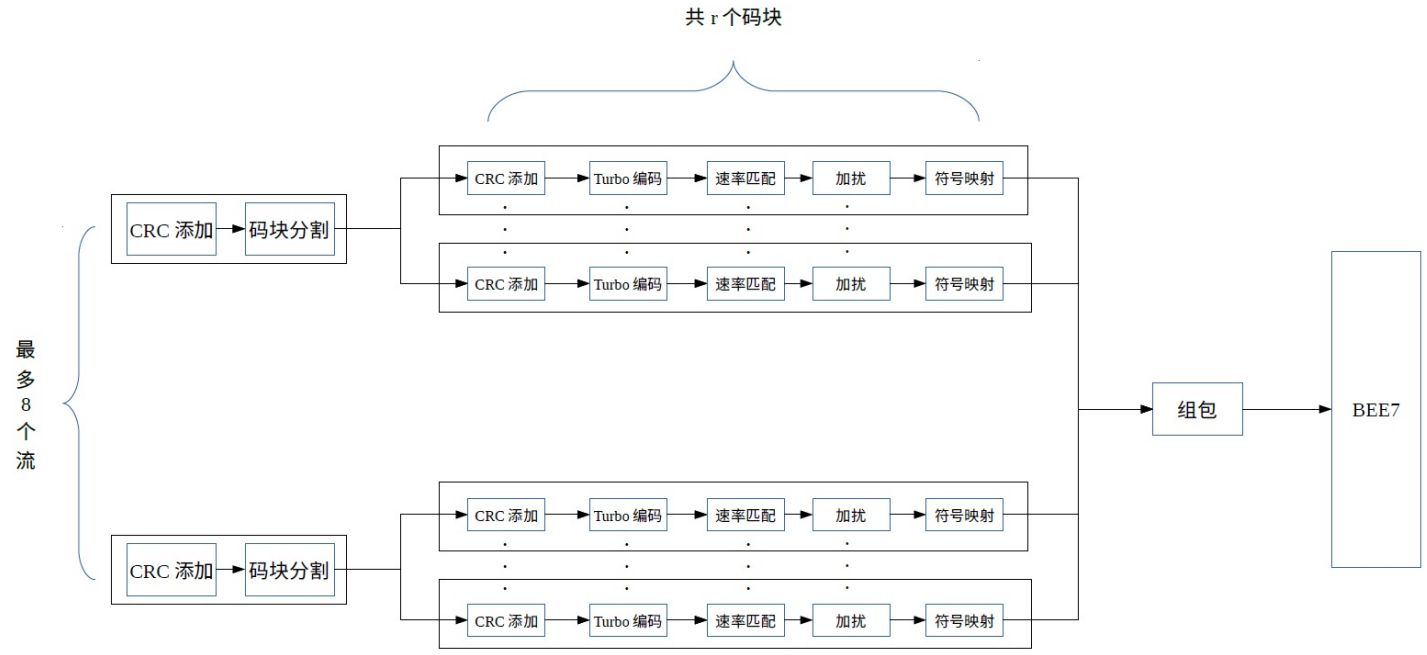

图 5: 发送端主线程流程图

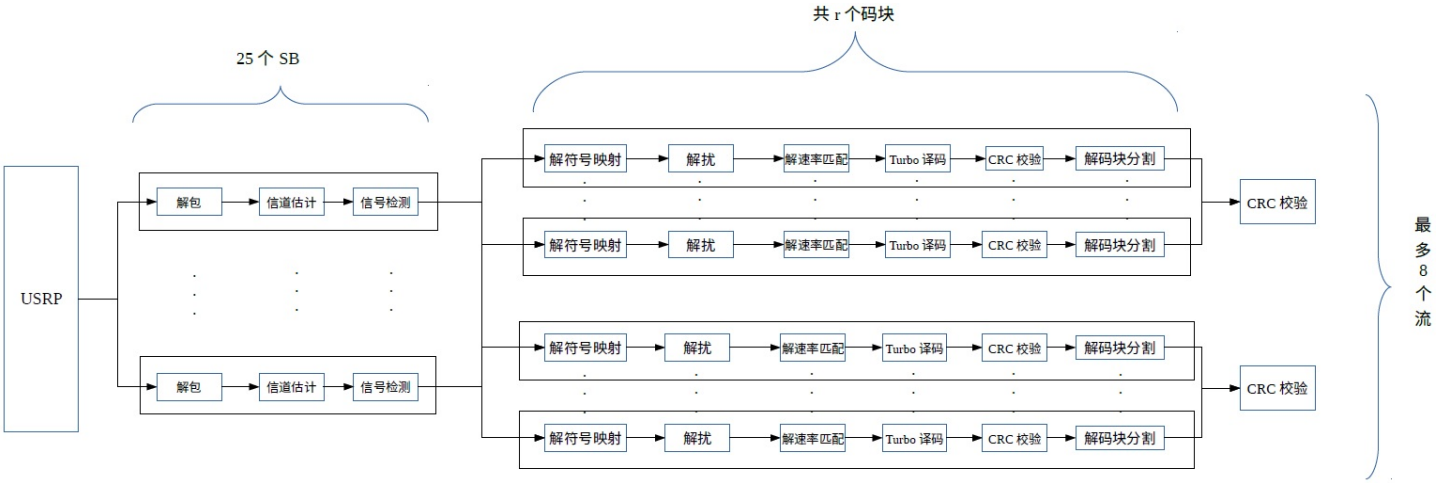

图 6: 接收端主线程流程图

4 存在问题

1. 每个收发主线程处理多少数据？一帧？分成八个流？
2. 为什么这样分割收发端主线程？是否越细分越好？
3. 收发端分离后通信的具体方式？DPDK？

5 下周计划

1. 继续阅读多线程部分源码
2. 学习 DPDK