第4讲 多线程Pthread编程

阅读内容

- 2.4 并行软件
  - 2.4.1 警告
  - 2.4.2 进程/线程协同
  - 2.4.3 共享内存
- 第四章 Pthreads共享内存编程

提纲

- 共享内存和分布式内存模型回顾
- POSIX Threads (Pthreads)编程简介
  - 基本概念
  - 基础API
  - 同步

提纲

- 共享内存和分布式内存模型回顾
- POSIX Threads (Pthreads)编程简介
  - 基本概念
  - 基础API
  - 同步

共享内存系统

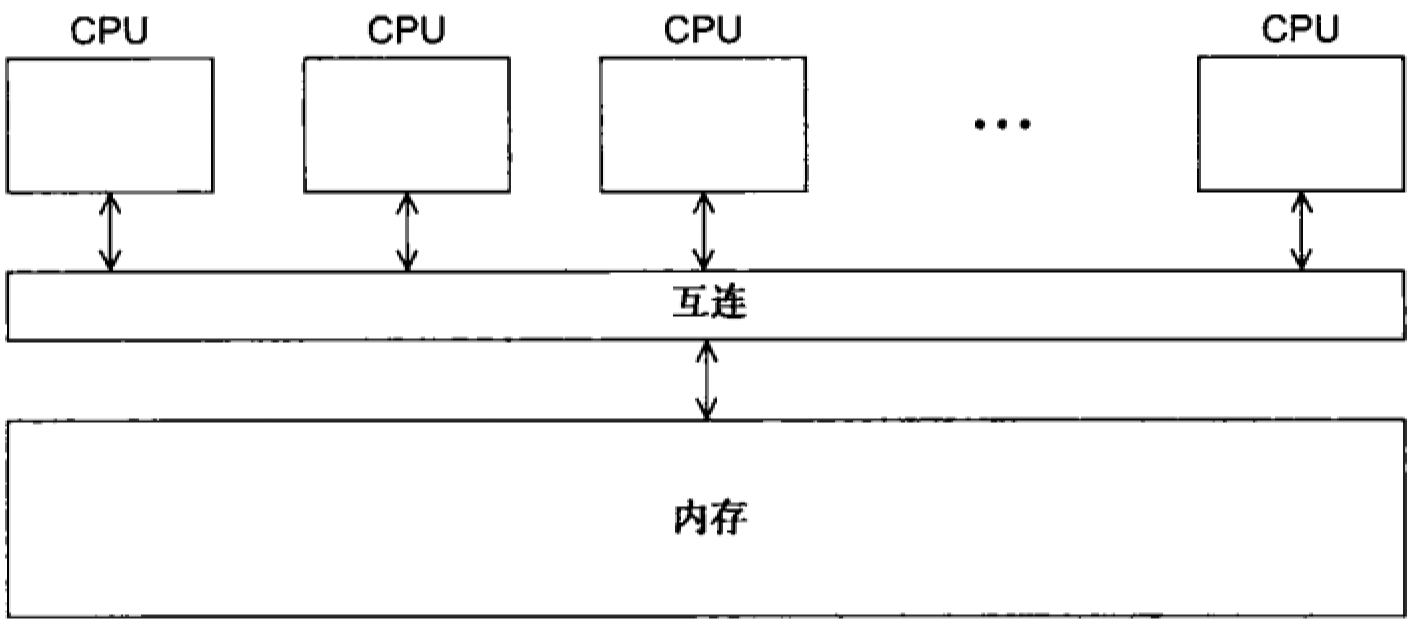


图 4-1　一个共享内存系统

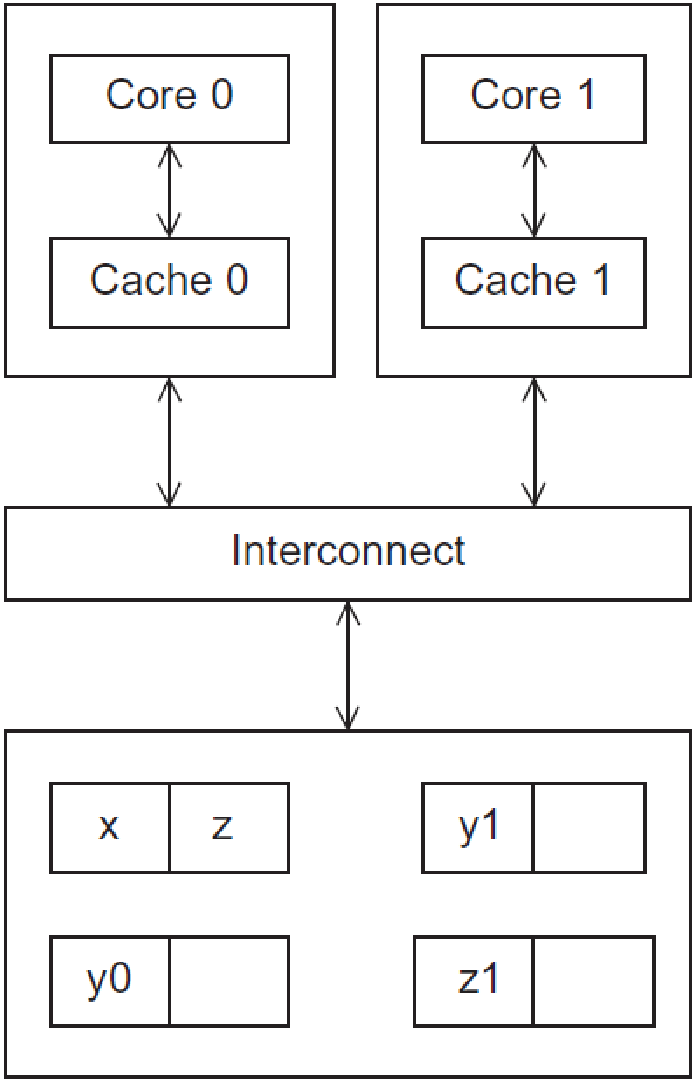
分类：

一致内存访问（UMA）多核系统；

非一致内存访问（NUMA）多核系统。

cache协同

- 程序员无法直接控制cache，也无法控制cache如何更新
- 但我们可以重新组织访存模式，来更好地利用cache



具有两个核心、分别有独享cache的共享内存系统

cache协同

y0是核心0私有的

y1和z1是核心1私有的

x = 2; /* 共享变量 */

Time	Core 0	Core 1
0	`y0 = x;`	`y1 = 3*x;`
1	`x = 7;`	Statement(s) not involving x
2	Statement(s) not involving x	`z1 = 4*x;`

y0 的最终结果 = 2

y1 的最终结果 = 6

z1 = ???

Cache一致性

- 基于侦听的cache协同
  - 核心共享总线;
  - 总线上传输的任何信号都能被连接到总线的所有核心“看到”；
- 基于目录的cache协同
  - 使用一种称为目录的数据结构保存每个cache line的状态
  - 当一个变量被更新时，就会查询目录，对缓存了该变量的核心，其缓存的副本的状态被置为无效

伪共享

- 一个cache line包含多个机器字
- 当多个处理器访问同一个cache line时，即使访问的是不同的机器字，看起来也有潜在的竞争条件
- 会产生不必要的协同开销

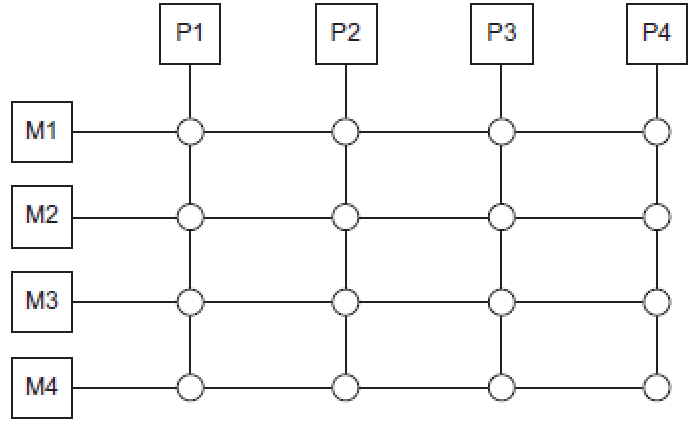
```
/* Core 0 does this */
for (i = 0; i < iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);

/* Core 1 does this */
for (i = iter_count+1; i < 2*iter_count; i++)
   for (j = 0; j < n; j++)
      y[i] += f(i,j);
```

共享内存互联

- 总线
  - 一组并行通信线路+总线访问控制硬件
  - 连接到总线的设备共享通信线路
  - 随着连接到总线的设备数增加，使用竞争就会加剧，性能会下降
- 交换互联
  - 使用交换开关控制数据在设备间的路由
  - 交叉开关Crossbar

(a)

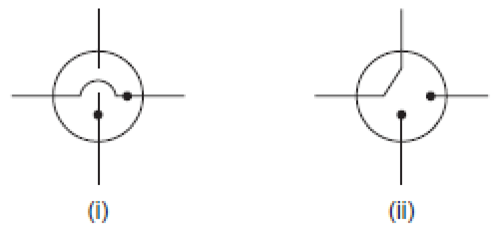
连接了四个处理器(P_i)和四个内存模块(M_j)的交叉开关



(a)

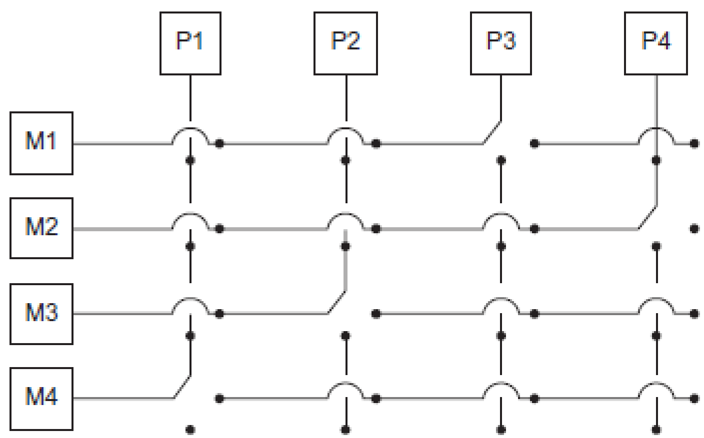
(b)

交换开关内部结构



(b)

(c) 不同处理器同时访问不同内存位置



(c)

共享内存编程

- 动态线程
  - 主线程等待计算工作，fork新线程分配工作，工作线程完成任务后结束
  - 资源高效利用，但线程创建/结束非常耗时
- 静态线程
  - 创建线程池，并向其中线程分配任务，但线程不结束，直至整个程序结束
  - 性能更优，但可能浪费系统资源

并行程序设计的复杂性

- 足够的并发度（Amdahl定律）
- 并发粒度
  - 独立的计算任务的大小
- 局部性
  - 对临近的数据进行计算
- 负载均衡
  - 处理器的工作量相近
- 协调和同步
  - 谁负责？处理频率？

线程安全/同步

- 第2章提到了共享内存并行函数或库的线程安全问题
  - 对一个函数或库，若多线程并行调用能“正确”执行，则称它是线程安全的
  - 由于多线程通过共享内存通信、协调，因此线程安全的代码使用恰当的同步操作修改共享内存状态
  - 某些串行代码的特性可能不是线程安全的？

性能评价

- 加速比
  - $S=T_S/T_P$
  - 阿姆达尔定律(Amdahl’s law)
- 效率
  - $E = S / p = T_S / (p*T_P)$
- 可扩展性
  - 增加程序核数(线程数/进程数)，如果在输入规模也以相应增长率增加的情况下，该程序的效率一直是E(不降)，则称该程序是可扩展的。

提纲

- 共享内存和分布式内存模型回顾
- POSIX Threads (Pthreads)编程简介
  - 基本概念
  - 基础API
  - 同步

多线程编程

已有多种线程库，还有更多的在开发中

- Pthread是POSIX标准
  - 相对低层
    - 程序员控制线程管理和协调
    - 程序员分解并行任务并管理任务调度
  - 可移植但可能较慢
  - 在系统级代码开发中广泛使用，也用于某些类型的应用程序
- OpenMP是新标准
  - 高层编程，适用于共享内存架构上的科学计算
    - 程序员在较高层次上指出并行方式和数据特性，并指导任务调度
    - 系统负责实际的并行任务分解和调度管理
  - 多种架构相关的编译指示

POSIX Thread(Pthread)概述

- POSIX：***P**ortable **O**perating **S**ystem **I**nterface for UNI**X***
  - 操作系统工具接口
- PThread：POSIX线程接口
  - 系统调用创建、同步线程
  - 应跨平台（类UNIX OS ）一致
- PThread支持
  - 创建并发执行
  - 同步
  - 非显式通信，因为共享内存是隐式的——共享数据的指针传递给线程

Slide source: Jim Demmel and Kathy Yelick

提纲

- 共享内存和分布式内存模型回顾
- POSIX Threads (Pthreads)编程简介
  - 基本概念
  - 基础API
  - 同步

Fork线程

API：

```
int pthread_create(pthread_t *,
                   const pthread_attr_t *,
                   void * (*)(void *),
                   void *);
```

调用例：

```
errcode = pthread_create(&thread_id,
                         &thread_attribute,
                          &thread_fun, &fun_arg);
```

- thread_id 是个指针，线程ID或句柄（用于停止线程等）
- thread_attribute 各种属性，通常用空指针NULL表示标准默认属性值
- thread_fun 新线程要运行的函数（参数和返回值类型都是void*）
- fun_arg 传递给要运行的函数thread_fun的参数
- errorcode 若创建失败，返回非零值

Slide source: Jim Demmel and Kathy Yelick

Fork线程

- pthread_create的效果
  - 主线程借助操作系统创建一个新线程
  - 线程执行一个特定函数thread_fun
  - 所有创建的线程执行相同的函数，表示线程的计算任务分解
  - 对于程序中不同线程执行不同任务的情况，可用创建线程时传递的参数区分线程的“id”以及其他线程的独特特性

简单的线程例子

```
int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, ParFun, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

编译：gcc … –lpthread

这段代码创建了16个线程执行函数“ParFun”.

注意：创建线程的代价很高，因此ParFun应完成很多工作才值得付出这种代价

Slide source: Jim Demmel and Kathy Yelick

线程数据共享

- 全局变量都是共享的
- 在堆中分配的对象可能是共享的（指针共享）
- 栈中的变量是私有的：将其指针传递给其他线程可能导致问题
- 常用共享方式：创建一个“线程数据”结构
  - 传递给所有线程，例如：
  -
```
char *message = "Hello World!\n";
    pthread_create( &thread1,
                    NULL,
                    (void*)&print_fun,
                    (void*) message);
```

Slide source: Jim Demmel and Kathy Yelick

Pthread “Hello World”程序

- 一些准备
  - 线程数（threadcount）运行时设置，从命令行读取
  - 每个线程打印“Hello from thread <X> of <threadcount>”
- 还需要另一个函数
  - **int pthread_join(pthread_t , void **value_ptr)**
  - UNIX说明：“挂起调用线程，直至目标线程结束，除非目标线程已结束。”
  - 第二个参数允许目标线程退出时返回信息给调用线程（通常是NULL）
  - 如发生错误返回非零值

Hello World程序

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* Global variable: accessible to all threads */
int thread_count;

void* Hello(void* rank); /* Thread function */

int main(int argc, char* argv[]) {
  long thread; /* Use long in case of a 64-bit system */
  pthread_t* thread_handles;

  /* Get number of threads from command line */
  thread_count = strtol(argv[1], NULL, 10);

  thread_handles = (pthread_t*) malloc(thread_count*sizeof(pthread_t));
```

各种Pthreads函数、常量、类型等的声明

线程执行的函数

线程句柄

从命令行读取线程数，或可改为直接赋值

Hello World程序(2)

```
for (thread = 0; thread < thread_count; thread++)
  pthread_create(&thread_handles[thread], NULL, Hello, (void*) thread);

 printf("Hello from the main thread\n");

 for (thread = 0; thread < thread_count; thread++)
   pthread_join(thread_handles[thread], NULL);

 free(thread_handles);
 return 0;
} /* main */
```

创建线程

线程ID

等待线程结束

Hello World程序(3)

```
void* Hello(void* rank) {
  long my_rank = (long) rank;  /* Use long in case of 64-bit system */

  printf("Hello from thread %ld of %d\n", my_rank, thread_count);

  return NULL;
} /* Hello */
```

- 可能的输出结果:

```
Hello from thread 0 of 8
Hello from thread 1 of 8
Hello from the main thread
Hello from thread 3 of 8
Hello from thread 4 of 8
Hello from thread 7 of 8
Hello from thread 2 of 8
Hello from thread 5 of 8
Hello from thread 6 of 8

Process returned 0 (0x0)   execution time : 0.037 s
Press any key to continue.
```

与你预想一样吗?

Pthread其他基础API

- void pthread_exit(void *value_ptr);
  - 通过value_ptr返回结果给调用者
- int pthread_cancel(pthread_t thread);
  - 取消线程thread执行

取消线程执行

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *threadFunc(void *parm)
{
		while(1)
		{
			fprintf(stdout, "I am the child thread.\n");
			pthread_testcancel();
			sleep(1);
		}
}
```

检测线程是否取消状态
若是，在此处退出线程
要注意什么？

取消线程执行（2）

```
int main(int argc, char *argv[])
{
        void        *status;
        pthread_t           thread;
        pthread_create(&thread, NULL, threadFunc, NULL);
        sleep(3);
        pthread_cancel(thread);
        pthread_join(thread, &status);
        if (status == PTHREAD_CANCELED)
                fprintf(stdout, "The child thread has been canceled.\n");
        else
                fprintf(stderr, "Unexpected thread status!\n");
        return 0;
}
```

向线程发出取消信号

等待线程真的退出

取消线程执行（3）

I am the child thread.
I am the child thread.
I am the child thread.
I am the child thread.
The child thread has been canceled.

提纲

- 共享内存和分布式内存模型回顾
- POSIX Threads (Pthreads)编程简介
  - 基本概念
  - 基础API
  - 同步

估算π

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++, factor = -factor)
  sum += factor/(2*k+1);
pi_approx = 4.0*sum;
```

多线程版本

```
void *pi_thread(void *parm)
{
  threadParm_t *p = (threadParm_t *) parm;
  int r = p->threadId;
  int n = p->n;
  int my_n = n/THREAD_NUM;
  int my_first = my_n*r;
  int my_last = my_first + my_n;

  if (my_first % 2 == 0) factor = 1.0;
  else factor = -1.0;

  for (int i = my_first; i < my_last; i++, factor = -factor) {
    sum += factor/(2*i+1);
  }
  pthread_exit(nullptr);
}
```

有什么问题?

多线程版本结果

- 串行版本运行结果

Seq: 3.1415826536, 0.85525284458ms.
Seq: 3.1415916536, 9.5339259821ms.
Seq: 3.1415925536, 96.029585625ms.
Seq: 3.1415926436, 975.36626866ms.

为什么结果不对?

- 四线程运行结果

Threaded: 3.333333333e-006, 0.9021998981ms.
Threaded: 3.3333333333e-007, 4.1811454101ms.
Threaded: 3.1415922536, 29.37293433ms.
Threaded: 3.1415926136, 251.96279507ms.

概念

- 原子性：一组操作要么全部执行要么全不执行，则称其是原子的。
- 临界区是一个更新共享资源的代码段，一次只能允许一个线程执行该代码段。
- 竞争条件：多个进程/线程尝试更新同一个共享资源时，结果可能是无法预测的，则存在竞争条件。（第2讲中还有其它描述方式）
- 数据依赖（data dependence）就是两个内存操作的序，为了保证结果的正确性，必须保持这个序
- 同步（synchronization）在时间上强制使各执行进程/线程在某一点必须互相等待，确保各进程/线程的正常顺序和对共享可写数据的正确访问。

同步——忙等待方法

```
void *pi_busywaiting(void *parm)
{
  threadParm_t *p = (threadParm_t *) parm;
  int r = p->threadId;   int n = p->n;   int my_n = n/THREAD_NUM;
  int my_first = my_n*r;   int my_last = my_first + my_n;
  double my_sum = 0.0;
  if (my_first % 2 == 0) factor = 1.0;
  else factor = -1.0;
  for (int i = my_first; i < my_last; i++, factor = -factor) {
    my_sum += factor/(2*i+1);
  }

  while (flag != r) Sleep(0);
  sum += my_sum;
  flag++;
  pthread_exit(nullptr);
}
```

flag指出的线程编号
才允许累加到全局和

避免过多忙等待
线程先各自求局部和

忙等待结果

- 串行版本运行结果

Seq: 3.1415826536, 0.85525284458ms.
Seq: 3.1415916536, 9.5339259821ms.
Seq: 3.1415925536, 96.029585625ms.
Seq: 3.1415926436, 975.36626866ms.

- 忙等待版本运行结果

Busy-Waiting: 3.1415826536, 8.8807495852ms.
Busy-Waiting: 3.1415916536, 4.9657735568ms.
Busy-Waiting: 3.1415925536, 34.058658506ms.
Busy-Waiting: 3.1415926436, 279.52112372ms.

显式同步：互斥量（锁）

- 创建mutex：

```
#include <pthread.h>
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init(&amutex, NULL);
```

- 使用mutex：

```
int pthread_mutex_lock(&amutex); // 加锁，若已上锁则阻塞至被解锁
int pthread_mutex_trylock(&amutex);
                                        // 加锁，若已上锁不阻塞，返回非0
int pthread_mutex_unlock(&amutex); // 解锁，可能令其他线程退出阻塞
```

- 释放mutex：

```
int pthread_mutex_destroy(&amutex);
```

幻灯片来源：Jim Demmel and Kathy Yelick

互斥量版本

```
void *pi_mutex(void *parm)
{
  threadParm_t *p = (threadParm_t *) parm;
  int r = p->threadId;   int n = p->n;  int my_n = n/THREAD_NUM;
  int my_first = my_n*r;   int my_last = my_first + my_n;
  double my_sum = 0.0;
  if (my_first % 2 == 0) factor = 1.0;
  else factor = -1.0;
  for (int i = my_first; i < my_last; i++, factor = -factor) {
    my_sum += factor/(2*i+1);
  }

  pthread_mutex_lock(&mutex);
  sum += my_sum;
  pthread_mutex_unlock(&mutex);
  pthread_exit(nullptr);
}
```

互斥量结果

- 忙等待版本运行结果

Busy-Waiting: 3.1415826536, 8.8807495852ms.
Busy-Waiting: 3.1415916536, 4.9657735568ms.
Busy-Waiting: 3.1415925536, 34.058658506ms.
Busy-Waiting: 3.1415926436, 279.52112372ms.

- 互斥量版本运行结果

Mutex: 3.1415826536, 0.91852930802ms.
Mutex: 3.1415916536, 5.4156488001ms.
Mutex: 3.1415925536, 27.708150988ms.
Mutex: 3.1415926436, 237.99910841ms.

*忙等待与互斥量区别：*进入临界区执行的线程顺序不同，前者指定顺序，后者顺序随机，指定顺序会造成资源浪费，先到的未必先执行；前者等待的线程也在空耗CPU计算资源，后者未进入临界区的线程会阻塞，释放CPU资源。

互斥量讨论

- 持有多个mutex可能导致死锁：

thread1	thread2
lock(a)	lock(b)
lock(b)	lock(a)

- 上锁/打开，二元状态

信号量

- 初始化信号量

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared, unsigned value);
```

Pshared 非0：进程间共享；0：进程内线程共享

value：信号量初始值

- 使用信号量：

```
int sem_wait(sem_t *sem); // 信号量值减1，若已为0则阻塞
int sem_post(sem_t *sem); // +1，若原来为0则可能唤醒阻塞线程
```

- 释放信号量：

```
int sem_destroy(sem_t *sem);
```

*注意：*信号量不是pthread库的一部分，需要加自己的头文件；有的系统，如MacOS不支持此信号量，但有类似的。

信号量同步例

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define  NUM_THREADS  4

typedef struct{
        int         threadId;
} threadParm_t;

sem_t    sem_parent;
sem_t    sem_children;
void *threadFunc(void *parm)
{
        threadParm_t     *p = (threadParm_t *) parm;
```

信号量同步例（2）

```
        fprintf(stdout, "I am the child thread %d.\n", p->threadId);
        sem_post(&sem_parent);
        sem_wait(&sem_children);
        fprintf(stdout, "Thread %d is going to exit.\n", p->threadId);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        sem_init(&sem_parent, 0, 0);
        sem_init(&sem_children, 0, 0);
        pthread_t        thread[NUM_THREADS];
        threadParm_t     threadParm[NUM_THREADS];
        int       i;
```

唤醒主线程——我已完成

等待主线程唤醒我

初始值都是0

信号量同步例（3）

```
for (i=0; i<NUM_THREADS; i++)
{
        threadParm[i].threadId = i;
        pthread_create(&thread[i], NULL, threadFunc, (void *)&threadParm[i]);
}

for (i=0; i<NUM_THREADS; i++)
{
        sem_wait(&sem_parent);  ← 等待所有子线程都输出
}

fprintf(stdout, "All the child threads has printed.\n");

for (i=0; i<NUM_THREADS; i++)
{
        sem_post(&sem_children);
}
```

唤醒子线程继续输出新内容

信号量同步例（4）

```
    for (i=0; i<NUM_THREADS; i++)
    {
            pthread_join(thread[i], NULL);
    }

    sem_destroy(&sem_parent);
    sem_destroy(&sem_children);
    return 0;
}
```

信号量同步例（5）

输出结果示例：

```
I am the child thread 0.
I am the child thread 1.
I am the child thread 2.
I am the child thread 3.
All the child threads has printed.
Thread 0 is going to exit.
Thread 3 is going to exit.
Thread 1 is going to exit.
Thread 2 is going to exit.
```

注意*： 信号量分两类：无名信号量（或未命名信号量）和命名信号量，以上介绍为无名信号量。MAC OSX 只支持命名信号量: sem_wait()、sem_post()仍然可用; 创建信号量 sem_init()要换成 sem_open(), 且信号量名字要以"/"开头 如"/mysem"; 删除命名信号量 sem_unlink(); 关闭命名信号量sem_close()。

使用barrier同步

- 初始化barrier的方法如下所示（本例中线程数为3）：

```
pthread_barrier_t b;
pthread_barrier_init(&b,NULL,3);
```

- 第二个参数指出对象属性，NULL表示默认属性
- 为等待barrier，线程应执行

```
pthread_barrier_wait(&b);
```

- 可通过下面的宏，指定一个初始值来初始化barrier

```
PTHREAD_BARRIER_INITIALIZER(3).
```

幻灯片来源：Jim Demmel and Kathy Yelick

barrier同步例

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define  NUM_THREADS  4

typedef struct{
        int         threadId;
}threadParm_t;

pthread_barrier_t barrier;

void *threadFunc(void *parm)
{
        threadParm_t      *p = (threadParm_t *) parm;
        fprintf(stdout, "Thread %d has entered step 1.\n", p->threadId);
        pthread_barrier_wait(&barrier);
```

barrier同步例（2）

```
        fprintf(stdout, "Thread %d has entered step 2.\n", p->threadId);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        pthread_barrier_init(&barrier, NULL, NUM_THREADS);
        pthread_t          thread[NUM_THREADS];
        threadParm_t       threadParm[NUM_THREADS];
        int        i;

        for (i=0; i<NUM_THREADS; i++)
        {
                threadParm[i].threadId = i;
                pthread_create(&thread[i], NULL, threadFunc, (void
*)&threadParm[i]);
        }
```

barrier同步例（3）

```
for (i=0; i<NUM_THREADS; i++)
{
        pthread_join(thread[i], NULL);
}

pthread_barrier_destroy(&barrier);
system("PAUSE");
return 0;
}
```

barrier同步例（4）

Thread 0 has entered step 1.
Thread 3 has entered step 1.
Thread 1 has entered step 1.
Thread 2 has entered step 1.
Thread 2 has entered step 2.
Thread 1 has entered step 2.
Thread 3 has entered step 2.
Thread 0 has entered step 2.

思考：用其他同步机制实现barrier？

补充：可以在init时指定n+1个等待，其中n是线程数。而在每个线程执行函数的首部调用wait()。这样100个pthread_create()结束后所有线程都停下来等待最后一个wait()函数被调用。这个wait()由主进程在它觉得合适的时候调用，相当于鸣响的起跑枪。

条件变量

- mutex是简单加锁/解锁
  - 如需判断条件后加锁，若条件不满足则需轮询，极耗资源
  - 使用条件变量，条件不满足时阻塞，在这之前会自动解锁被唤醒后自动加锁，再去检测条件
- API

```
pthread_cond_init (condition,attr)
pthread_cond_destroy (condition)
pthread_condattr_init (attr)
pthread_condattr_destroy (attr)
pthread_cond_wait (condition,mutex) // 条件不成立便阻塞，之前解锁
pthread_cond_signal (condition)      // 触发条件，可能唤醒一个阻塞线程
pthread_cond_broadcast (condition) // 唤醒多个线程
```

条件变量例

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12

int     count = 0;
int     thread_ids[3] = {0,1,2};
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;
```

两个线程并发递增同一个计数器
一个观察线程等待计数器达到12

条件变量例（2）

```
int main (int argc, char *argv[])
{
  int i, rc;
  pthread_t threads[3];
  pthread_attr_t attr;

  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, inc_count, (void *)&thread_ids[0]);
  pthread_create(&threads[1], &attr, inc_count, (void *)&thread_ids[1]);
  pthread_create(&threads[2], &attr, watch_count, (void *)&thread_ids[2]);
```

条件变量例（3）

```
  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);
}
```

条件变量例（4）

```
void *inc_count(void *idp)    计数线程
{
  int j,i;
  double result=0.0;
  int *my_id = idp;

  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;      递增计数器

    if (count == COUNT_LIMIT) {          唤醒等待在条件变量上
      pthread_cond_signal(&count_threshold_cv);   的观察线程
      printf("inc_count(): thread %d, count = %d  Threshold
              reached.\n",  *my_id, count);
   }
    printf("inc_count(): thread %d, count = %d, unlocking mutex\n",
                                        *my_id, count);
    pthread_mutex_unlock(&count_mutex);
```

条件变量例（5）

```
    for (j=0; j < 1000; j++)
      result = result + (double)rand();
  }
  pthread_exit(NULL);
}
```

条件变量例（6）

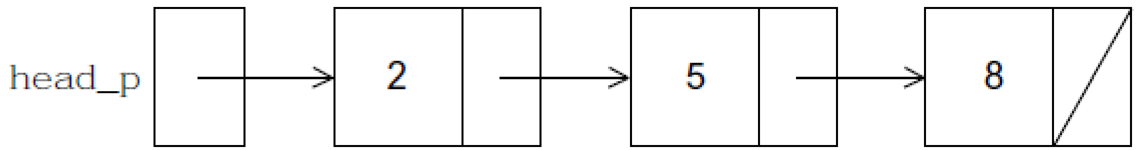
```
void *watch_count(void *idp)   观察线程
{
  int *my_id = idp;

  printf("Starting watch_count(): thread %d\n", *my_id);

  pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch_count(): thread %d Condition signal received.\n", *my_id);
  }
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

等待条件变量，睡眠之前解锁互斥量

条件变量（7）

```
inc_count(): thread 0, count = 1, unlocking mutex
Starting watch_count(): thread 2
inc_count(): thread 1, count = 2, unlocking mutex
inc_count(): thread 0, count = 3, unlocking mutex
inc_count(): thread 1, count = 4, unlocking mutex
inc_count(): thread 0, count = 5, unlocking mutex
inc_count(): thread 1, count = 6, unlocking mutex
inc_count(): thread 0, count = 7, unlocking mutex
inc_count(): thread 1, count = 8, unlocking mutex
inc_count(): thread 0, count = 9, unlocking mutex
inc_count(): thread 1, count = 10, unlocking mutex
inc_count(): thread 0, count = 11, unlocking mutex
inc_count(): thread 1, count = 12  Threshold reached.
inc_count(): thread 1, count = 12, unlocking mutex
watch_count(): thread 2 Condition signal received.
inc_count(): thread 0, count = 13, unlocking mutex
inc_count(): thread 1, count = 14, unlocking mutex
inc_count(): thread 0, count = 15, unlocking mutex
inc_count(): thread 1, count = 16, unlocking mutex
inc_count(): thread 0, count = 17, unlocking mutex
inc_count(): thread 1, count = 18, unlocking mutex
inc_count(): thread 0, count = 19, unlocking mutex
inc_count(): thread 1, count = 20, unlocking mutex
Main(): Waited on 3  threads. Done.
```

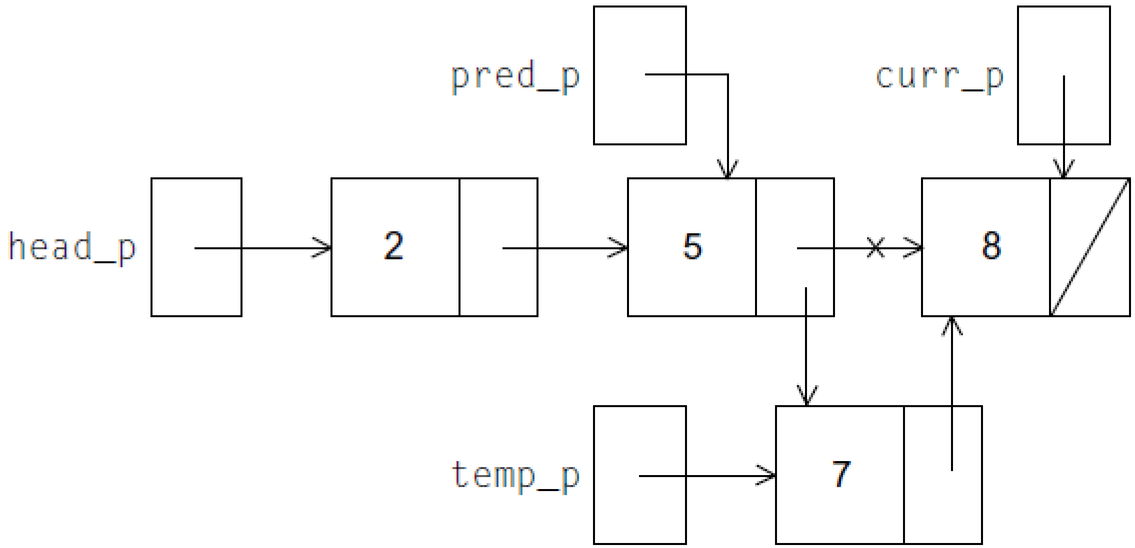
读写锁：链表函数




```c
int  Member(int value, struct list_node_s* head_p) {
   struct list_node_s* curr_p = head_p;

   while (curr_p != NULL && curr_p->data < value)
      curr_p = curr_p->next;

   if (curr_p == NULL || curr_p->data > value) {
      return 0;
   } else {
      return 1;
   }
}  /* Member */
```

在升序链表中查找给定值

Insert函数



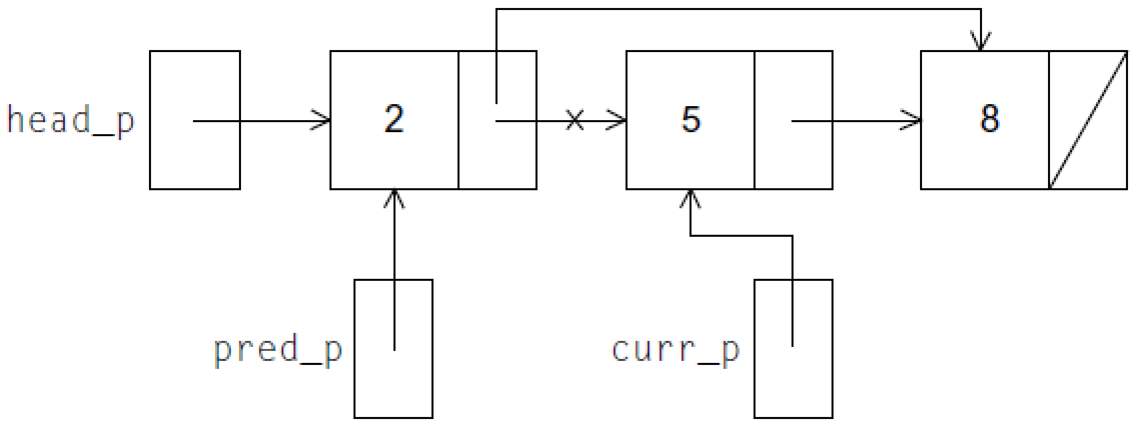
Insert函数

```c
int Insert(int value, struct list_node_s** head_p) {
   struct list_node_s* curr_p = *head_p;
   struct list_node_s* pred_p = NULL;
   struct list_node_s* temp_p;

   while (curr_p != NULL && curr_p->data < value) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }

   if (curr_p == NULL || curr_p->data > value) {
      temp_p = malloc(sizeof(struct list_node_s));
      temp_p->data = value;
      temp_p->next = curr_p;
      if (pred_p == NULL)  /* New first node */
         *head_p = temp_p;
      else
         pred_p->next = temp_p;
      return 1;
   } else { /* Value already in list */
      return 0;
   }
}  /* Insert */
```

查找插入位置

修改链表中指针

Delete函数



Delete函数

```c
int Delete(int value, struct list_node_s** head_p) {
   struct list_node_s* curr_p = *head_p;
   struct list_node_s* pred_p = NULL;

   while (curr_p != NULL && curr_p->data < value) {
      pred_p = curr_p;
      curr_p = curr_p->next;
   }

   if (curr_p != NULL && curr_p->data == value) {
      if (pred_p == NULL) { /* Deleting first node in list */
         *head_p = curr_p->next;
         free(curr_p);
      } else {
         pred_p->next = curr_p->next;
         free(curr_p);
      }
      return 1;
   } else { /* Value isn't in list */
      return 0;
   }
} /* Delete */
```

查找删除元素

修改链表中指针

对整个链表加锁

```
Pthread_mutex_lock(&list_mutex);
Member(value);
Pthread_mutex_unlock(&list_mutex);
```

粒度太粗，可能令线程不必要地等待

结点级锁

```
struct list_node_s {
   int data;
   struct list_node_s* next;
   pthread_mutex_t mutex;
}
```

```c
int  Member(int value) {
   struct list_node_s* temp_p;

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;
   while (temp_p != NULL && temp_p->data < value) {
      if (temp_p->next != NULL)
         pthread_mutex_lock(&(temp_p->next->mutex));
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      temp_p = temp_p->next;
   }

   if (temp_p == NULL || temp_p->data > value) {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      if (temp_p != NULL)
         pthread_mutex_unlock(&(temp_p->mutex));
      return 0;
   } else {
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      return 1;
   }
}  /* Member */
```

Pthread读写锁

```
pthread_rwlock_rdlock(&rwlock);
Member(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);
. . .
pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```

```
int pthread_rwlock_init(
        pthread_rwlock_t*             rwlock_p   /* out */,
        const pthread_rwlockattr_t*   attr_p     /* in  */);
```

```
int pthread_rwlock_destroy(pthread_rwlock_t* rwlock_p /* in/out */);
```

性能对比

Table 4.3 Linked List Times: 1000 Initial Keys, 100,000 ops, 99.9% `Member`, 0.05% `Insert`, 0.05% `Delete`

Implementation	Number of Threads			
	1	**2**	**4**	**8**
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

Table 4.4 Linked List Times: 1000 Initial Keys, 100,000 ops, 80% `Member`, 10% `Insert`, 10% `Delete`

Implementation	Number of Threads			
	1	**2**	**4**	**8**
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

综合例：多个数组排序

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <time.h>
#include <immintrin.h>
#include <windows.h>
#include <pthread.h>

using namespace std;

typedef struct{
        int         threadId;
} threadParm_t;

const int ARR_NUM = 10000;
const int ARR_LEN = 10000;
const int THREAD_NUM = 4;
const int seg = ARR_NUM / THREAD_NUM;
```

多个一维数组也可看作一个矩阵

对每行（一维数组）进行排序
与矩阵与向量相乘有何差别？

综合例：多个数组排序（2）

```
vector<int> arr[ARR_NUM];
pthread_mutex_t  mutex;
long long head, freq;        // timers

void init(void)
{
  srand(unsigned(time(nullptr)));
  for (int i = 0; i < ARR_NUM; i++) {
    arr[i].resize(ARR_LEN);
    for (int j = 0; j < ARR_LEN; j++)
      arr[i][j] = rand();
  }
}
```

综合例：多个数组排序（3）

```
void *arr_sort(void *parm)
{
  threadParm_t *p = (threadParm_t *) parm;
  int r = p->threadId;
  long long tail;

  for (int i = r * seg; i < (r + 1) * seg; i++)
    sort(arr[i].begin(), arr[i].end());

  pthread_mutex_lock(&mutex);
  QueryPerformanceCounter((LARGE_INTEGER *)&tail);
  printf(“Thread %d: %lfms.\n", r, (tail - head) * 1000.0 / freq);
  pthread_mutex_unlock(&mutex);

  pthread_exit(nullptr);
}
```

每个线程负责连续 n/4个数组的排序

一种数据划分方法——块划分

综合例：多个数组排序（4）

```
int main(int argc, char *argv[])
{
 QueryPerformanceFrequency((LARGE_INTEGER *)&freq);

 init();
 mutex = PTHREAD_MUTEX_INITIALIZER;
 pthread_t thread[THREAD_NUM];
 threadParm_t threadParm[THREAD_NUM];

 QueryPerformanceCounter((LARGE_INTEGER *)&head);

 for (int i = 0; i < THREAD_NUM; i++)
 {
  threadParm[i].threadId = i;
  pthread_create(&thread[i], nullptr, arr_sort, (void *)&threadParm[i]);
 }
```

综合例：多个数组排序（5）

```
  for (int i = 0; i < THREAD_NUM; i++)
  {
    pthread_join(thread[i], nullptr);
  }

  pthread_mutex_destroy(&mutex);
}
```

综合例：多个数组排序（6）

- 单线程

Thread 0: 7581.931894ms.

- 4线程

Thread 3: 1942.302817ms.
Thread 2: 1948.374916ms.
Thread 0: 1955.479851ms.
Thread 1: 1969.761978ms.

有什么问题?

综合例：多个数组排序（7）

```
void init_2(void)
{
  int ratio;
  srand(unsigned(time(nullptr)));
  for (int i = 0; i < ARR_NUM; i++) {
    arr[i].resize(ARR_LEN);
    if (i < seg) ratio = 0;
    else if (i < seg * 2) ratio = 32;
    else if (i < seg * 3) ratio = 64;
    else ratio = 128;
    if ((rand() & 127) < ratio)
      for (int j = 0; j < ARR_LEN; j++)
        arr[i][j] = ARR_LEN - j;
    else
      for (int j = 0; j < ARR_LEN; j++)
        arr[i][j] = j;
  }
}
```

前1/4：完全升序
第二段：1/4逆序，3/4升序
第三段：1/2逆序，1/2升序
第四段：完全逆序
块划分负载不均！

综合例：多个数组排序（8）

- 单线程

Thread 0: 1643.106837ms.

- 4线程

Thread 0: 428.869616ms.
Thread 1: 486.402280ms.
Thread 2: 530.073299ms.
Thread 3: 643.510582ms.

如何解决?

综合例：多个数组排序（9）

```
int next_arr = 0;
pthread_mutex_t  mutex_task;
void *arr_sort_fine(void *parm)
{
  threadParm_t *p = (threadParm_t *) parm;
  int r = p->threadId;
  int task = 0;
  long long tail;
  while (1) {
    pthread_mutex_lock(&mutex_task);
    task = next_arr++;
    pthread_mutex_unlock(&mutex_task);
    if (task >= ARR_NUM) break;
    stable_sort(arr[task].begin(), arr[task].end());
  }
  pthread_mutex_lock(&mutex);
  QueryPerformanceCounter((LARGE_INTEGER *)&tail);
  printf("Thread %d: %lfms.\n", r, (tail - head) * 1000.0 / freq);
  pthread_mutex_unlock(&mutex);
  pthread_exit(nullptr);
}
```

获取任务
——动态任务划分

综合例：多个数组排序（10）

- 动态任务划分

Thread 0: 549.246907ms.
Thread 3: 552.934092ms.
Thread 2: 556.541263ms.
Thread 1: 559.427082ms.

继续改进？

- 粗粒度动态划分——每次分配50行

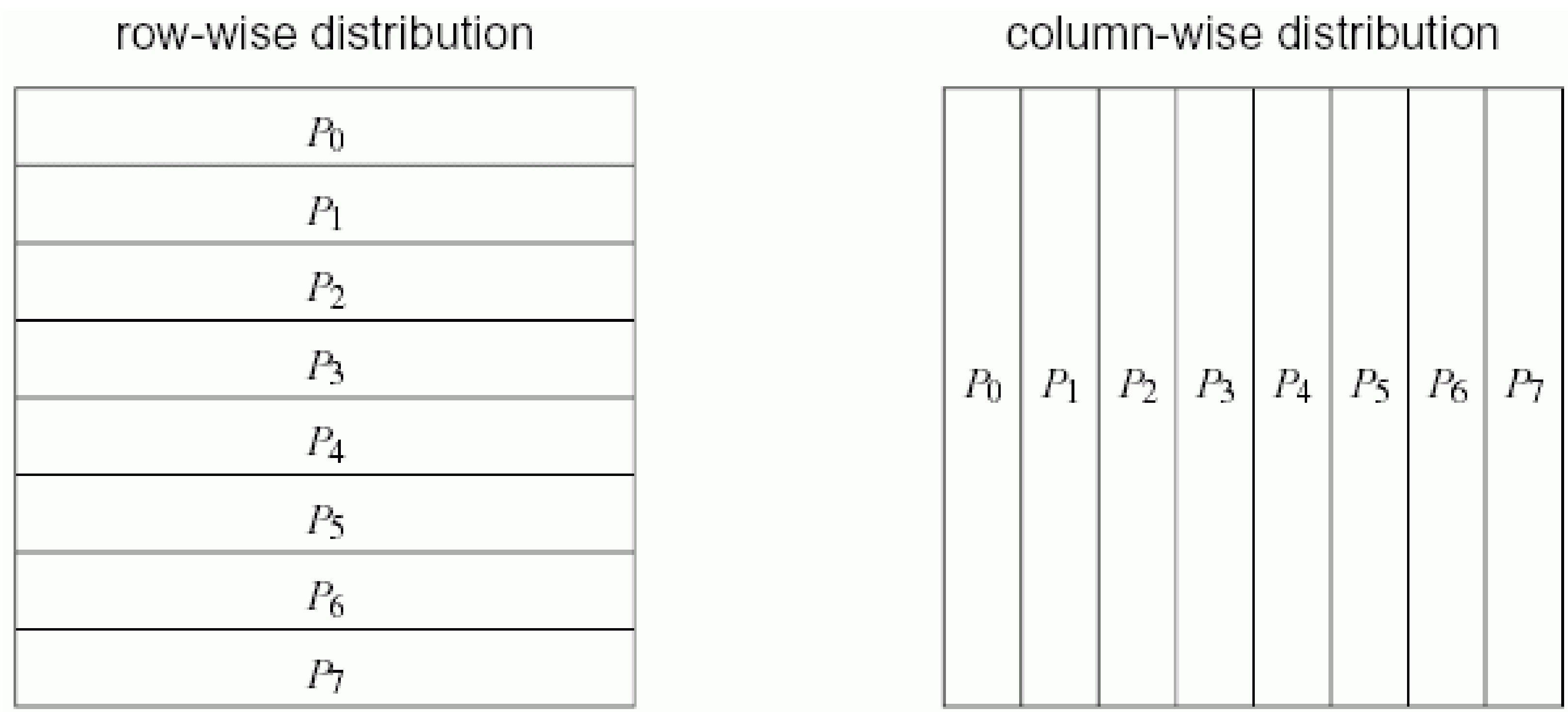
Thread 0: 520.849620ms.
Thread 1: 524.470671ms.
Thread 3: 527.458957ms.
Thread 2: 530.890995ms.

Pthread编程小结

- Pthread是基于OS特性的
  - 可用于多种语言（需要适合的头文件）
  - 支持的语言是大多数程序员所熟悉的
  - 数据共享很方便
- 缺点
  - 数据竞争很难发现
- 当前，程序员常用更简单的OpenMP，当然会有一些限制
  - 用少量编译指示指出并行任务和共享数据，即可将串行程序多线程化

数据划分方法

- 块划分（一维）：连续n/p行（列）



二维块划分

P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}

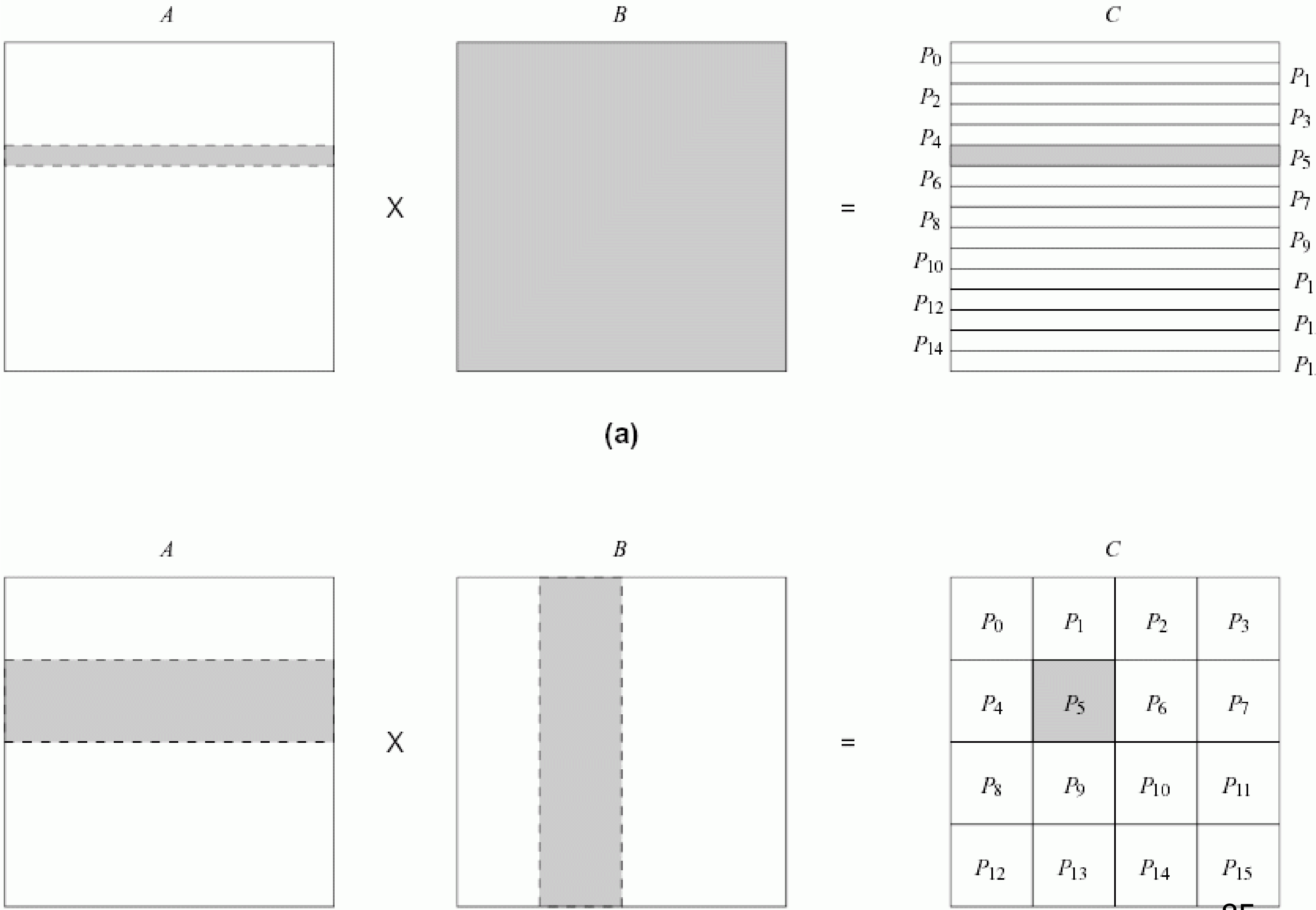
高维划分减少交互

通信量

n^2+n^2/p

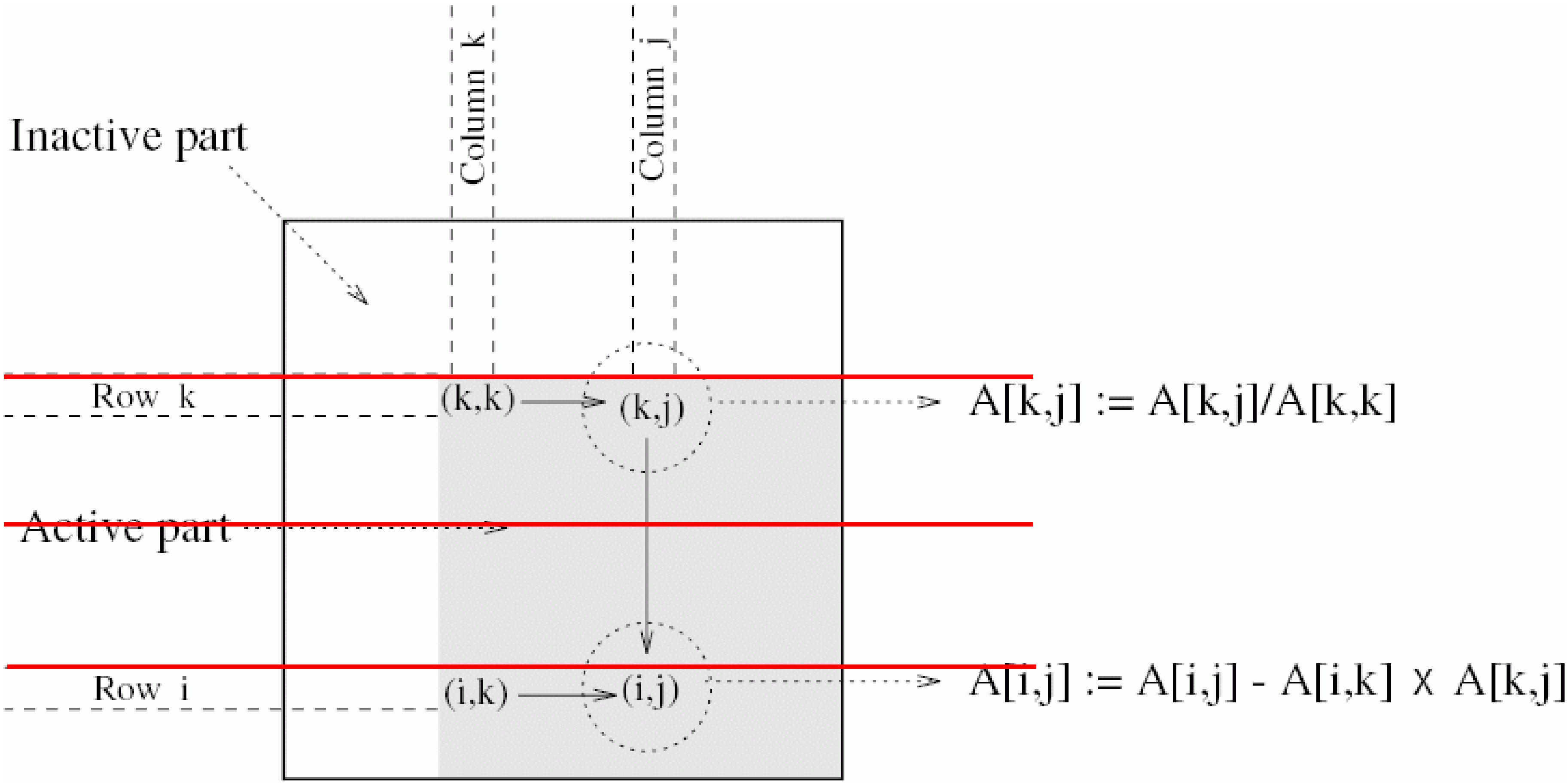
通信量

$2n^2/\sqrt{p}$



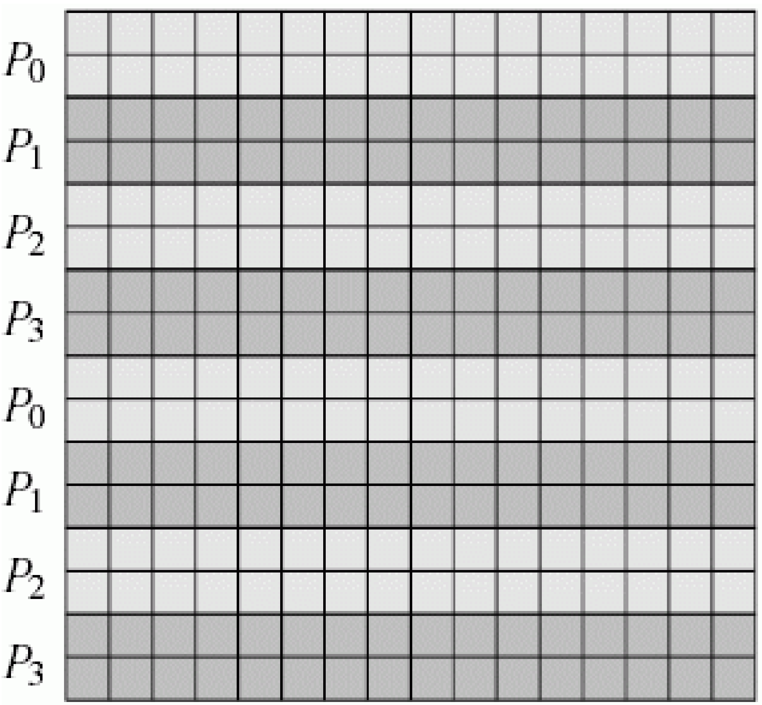
多线程高斯消去

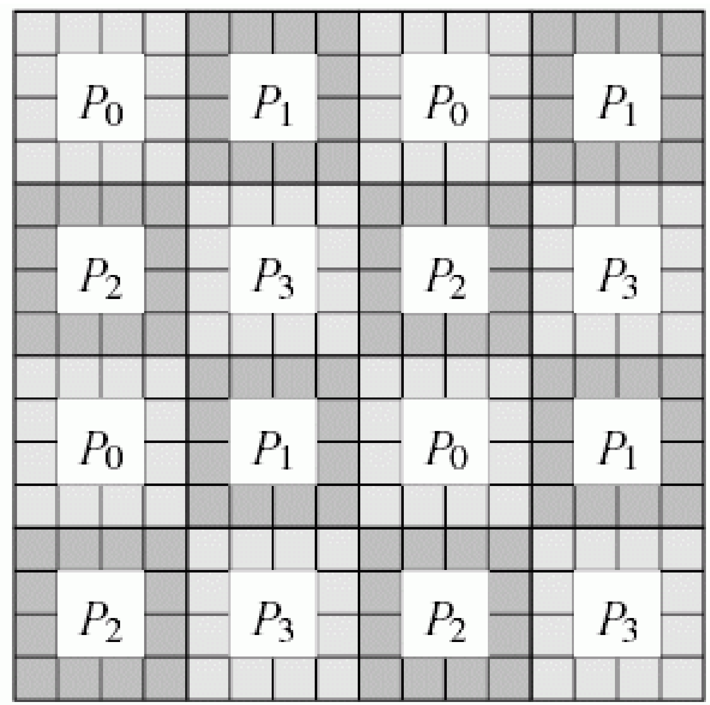
- 难点？**块划分负载不均！**



（块）循环划分

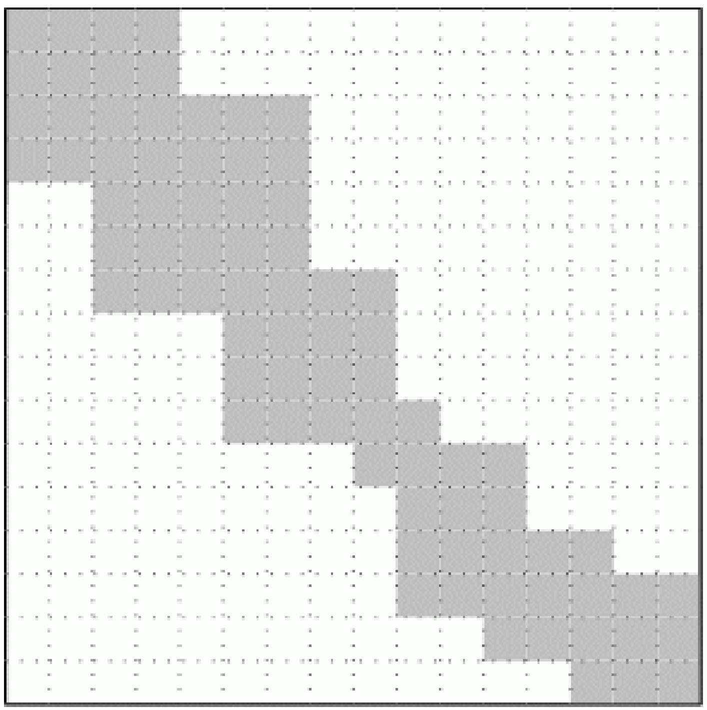
- 任务数>线程数，循环分配给线程
- 每个线程负责的区域散布在整个矩阵中，负载不均大大缓解





随机块划分

- 计算量分布无任何规律——稀疏矩阵
- 循环分配也无法保证负载均衡



P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}
P_0	P_1	P_2	P_3	P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}	P_{12}	P_{13}	P_{14}	P_{15}

随机块划分

- 每个维度随机排列
- 每个线程仍划分一个连续区域