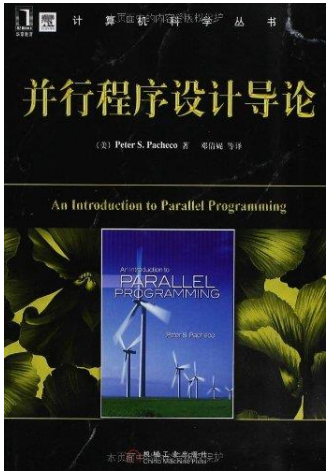


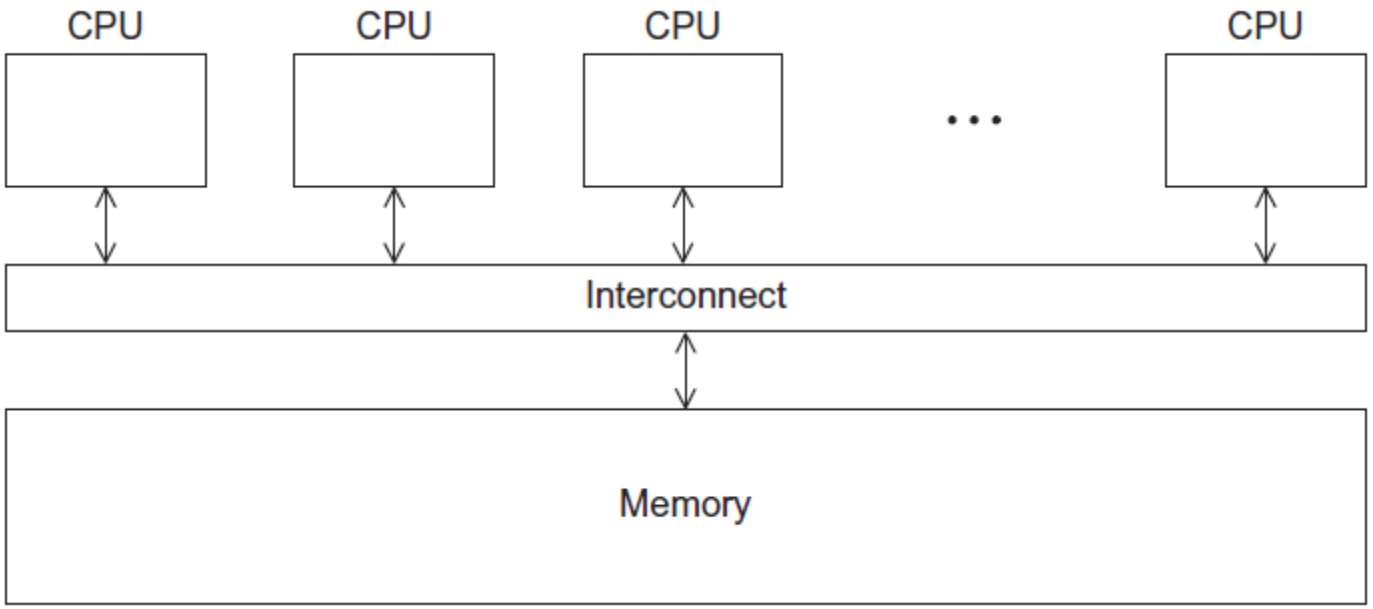
第四章

用Pthreads进行共享内存编程

目录

- 共享内存系统编程的问题.
- 临界区的控制访问
- 线程同步
- 使用POSIX threads编程.
- 互斥量
- 生产-消费者同步和信号量
- 路障和条件变量
- 读写锁
- 线程安全

共享存储系统



进程和线程

- 进程是正在运行(或挂起)的程序的实例.
- 线程类似于“轻量级”进程。
- 在共享内存程序中，单个进程可以有多个控制线程.

POSIX®Threads

- 也被称为Pthreads
- 类Unix操作系统的标准
- 一个可以与C程序链接的库
- 多线程编程的应用程序编程接口(API)

注意

- Pthreads 的API只有在支持POSIXR的系统上才有效— Linux, MacOS X, Solaris, HPUX, …

Hello World! (1)

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

声明各种Pthreads函数、常量、类型等.

Hello World! (2)

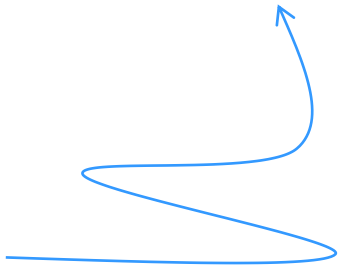
```
    for (thread = 0; thread < thread_count; thread++)
       pthread_create(&thread_handles[thread], NULL,
           Hello, (void*) thread);

    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 */
```

Hello World! (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 */
```

编译Pthread程序

gcc −g −Wall −o pth_hello pth_hello . c −lpthread



运行Pthreads程序

. / pth_hello <number of threads>

. / pth_hello 1

Hello from the main thread
Hello from thread 0 of 1

. / pth_hello 4

Hello from the main thread
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

全局变量

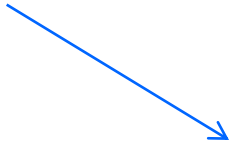
- 全局变量可能在程序中引发令人困惑的错误
- 限制使用全局变量，除了确实需要用到的情况外。
  - 线程之间的共享变量

启动线程

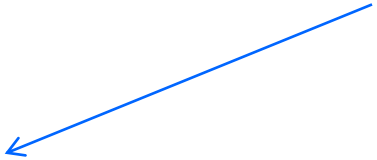
- MPI中的进程通常由脚本启动。
- 在Pthreads中，线程由程序可执行文件启动

启动线程

pthread.h



pthread_t



One object for each thread.

```
int pthread_create (
        pthread_t*  thread_p /* out */ ,
        const pthread_attr_t*  attr_p /* in */ ,
        void*  (*start_routine ) ( void ) /* in */ ,
        void*  arg_p /* in */ ) ;
```

pthread_t 对象

- 不透明对象
- 对象中存储的数据都是系统绑定的
- 用户级代码不能直接访问它们的数据成员
- 然而，Pthreads标准保证pthread_t对象中必须存储足够多的信息，以唯一地标识与它所从属的线程。

pthread_create(1)

```
int pthread_create (
      pthread_t*  thread_p /* out */ ,
      const pthread_attr_t*  attr_p /* in */ ,
      void*  (*start_routine ) ( void ) /* in */ ,
      void*  arg_p /* in */ ) ;
```

We won't be using, so we just pass NULL.

Allocate before calling.

pthread_create(2)

```
int pthread_create (
        pthread_t*  thread_p /* out */ ,
        const pthread_attr_t*  attr_p /* in */ ,
        void*  (*start_routine ) ( void ) /* in */ ,
        void*  arg_p /* in */ ) ;
```

Pointer to the argument that should be passed to the function *start_routine*.

The function that the thread is to run.

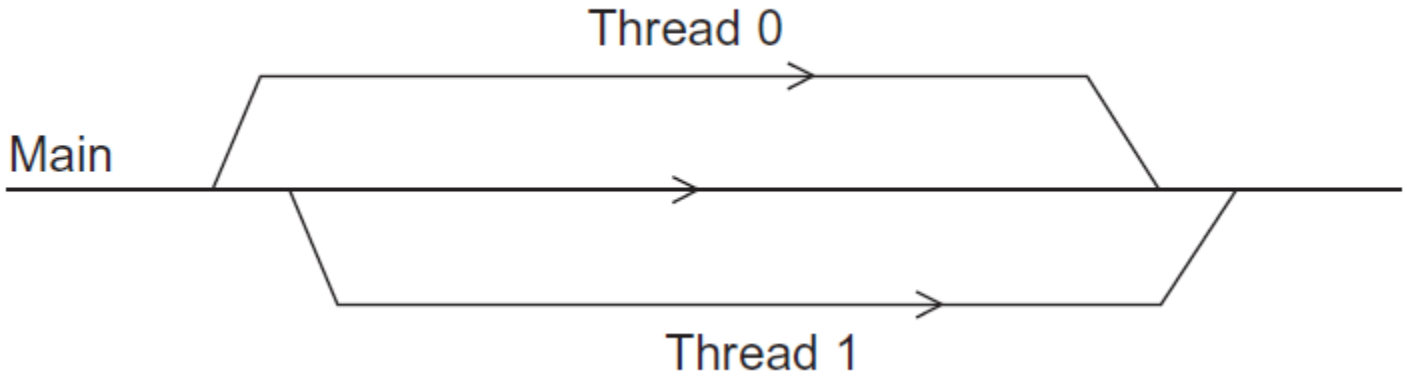
pthread_create生成并运行的函数

- 原型:
  void* thread_function (void* args_p) ;

- Void* 可以转换为C语言中的任意指针类型.

- 因此，args_p可以指向一个列表，该列表包含一个或多个thread_function需要的值。
- 类似地，thread_function的返回值可以是包含一个或多个值的列表

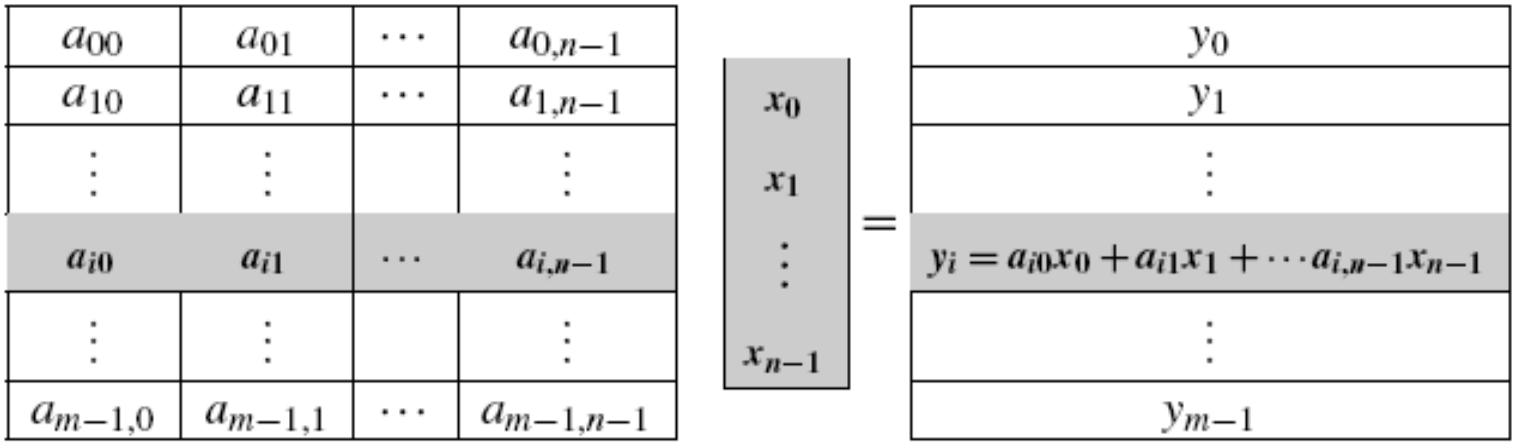
运行线程



主线程派生与合并两个线程

停止线程

- 对每个线程调用一次pthread_join函数。
- 调用一次pthread_join函数将等待pthread_t对象所关联的那个线程结束



PTHREADS中的矩阵-向量乘法

串行伪代码

```
/* For each row of A */
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /* For each element of the row and each element of x */
    for (j = 0; j < n; j++)
        y[i] += A[i][j]* x[j];
}
```

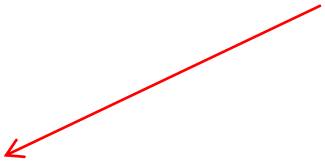
$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j$$

Using 3 Pthreads

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

thread 0

```
y[0] = 0.0;
for (j = 0; j < n; j++)
   y[0] += A[0][j]* x[j];
```



general case

```
y[i] = 0.0;
for (j = 0; j < n; j++)
   y[i] += A[i][j]*x[j];
```

Pthreads矩阵-向量乘法

```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
   }

   return NULL;
} /* Pth_mat_vect */
```

临界区

估算 π

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
    sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

计算π的线程函数

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)  /* my_first_i is even */
      factor = 1.0;
   else  /* my_first_i is odd */
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      sum += factor/(2*i+1);
   }

   return NULL;
}  /* Thread_sum */
```

使用双核处理器

	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

请注意，随着n的增加，单线程的估计会越来越准确

可能的竞争条件

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call `Compute()`	Started by main thread
3	Assign `y = 1`	Call `Compute()`
4	Put `x=0` and `y=1` into registers	Assign `y = 2`
5	Add 0 and 1	Put `x=0` and `y=2` into registers
6	Store 1 in memory location x	Add 0 and 2
7		Store 2 in memory location x

忙等待

- 在忙等待中，线程不停地测试某个条件，但实际上，直到某个条件满足之前，不会执行任何有用的工作.

- 但是要注意编译器的优化!会影响忙等待的正确执行

```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

flag initialized to 0 by main thread

忙等待求全局和的Pthreads程序

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

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

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      while (flag != my_rank);
      sum += factor/(2*i+1);
      flag = (flag+1) % thread_count;
   }

   return NULL;
}  /* Thread_sum */
```

循环后用临界区求全局和的函数

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor, my_sum = 0.0;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

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

   for (i = my_first_i; i < my_last_i; i++, factor = -factor)
      my_sum += factor/(2*i+1);

   while (flag != my_rank);
   sum += my_sum;
   flag = (flag+1) % thread_count;

   return NULL;
}  /* Thread_sum */
```

互斥量（ Mutex ）

- 一个处于忙等待状态的线程任然会持续地使用CPU，而什么也没有完成。

- 互斥量(互斥锁)是一种特殊类型的变量，它可以限制每次只允许一个线程访问临界区。

互斥量

- 用于保证一个线程独享临界区，其它线程在有线程已经进入该临界区的情况下，不能同时进入。
- Pthreads标准为互斥提供了一种特殊的类型:pthread_mutex_t。

```
int pthread_mutex_init(
      pthread_mutex_t*              mutex_p    /* out */
      const pthread_mutexattr_t*    attr_p     /* in  */);
```

互斥量

- 当Pthreads程序使用完互斥量后，它应该调用

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p  /* in/out */);
```

- 为了获得对临界区的访问，线程调用

```
int pthread_mutex_lock(pthread_mutex_t* mutex_p  /* in/out */);
```

- 当一个线程退出临界区后，它应该调用：

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p  /* in/out */);
```

用互斥量计算全局和

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;
   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      my_sum += factor/(2*i+1);
   }
   pthread_mutex_lock(&mutex);
   sum += my_sum;
   pthread_mutex_unlock(&mutex);

   return NULL;
} /* Thread_sum */
```

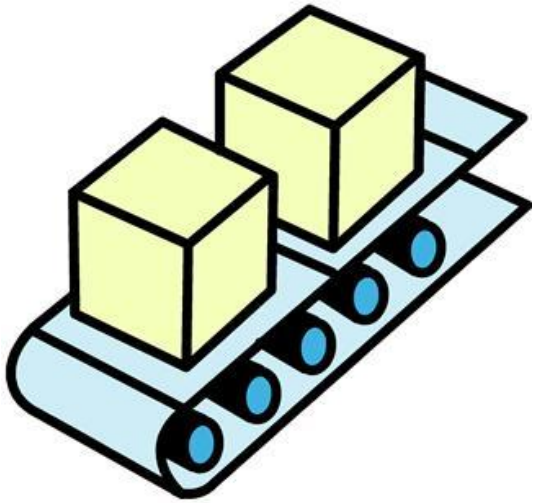
Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

$$\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \texttt{thread\_count}$$

计算π的程序，使用n=10^8个项目，在一个有两个4核处理器的系统上运行时间（秒）

Time	flag	Thread				
		0	1	2	3	4
0	0	crit sect	busy wait	susp	susp	susp
1	1	terminate	crit sect	susp	busy wait	susp
2	2	—	terminate	susp	busy wait	busy wait
⋮	⋮			⋮	⋮	⋮
?	2	—	—	crit sect	susp	busy wait

采用忙等待，并行线程个数多于核的个数时，可能的线程执行顺序



生产者-消费者同步和信号量

问题

- 忙等待强制顺序线程访问临界区，造成资源浪费
- 使用互斥量，顺序由系统自己决定。
- 在一些应用程序中，我们需要控制线程访问临界区的顺序。

互斥量解决方案存在的问题

```
/* n and product_matrix are shared and initialized by the main thread */
/* product_matrix is initialized to be the identity matrix            */
void* Thread_work(void* rank) {
   long my_rank = (long) rank;
   matrix_t my_mat = Allocate_matrix(n);
   Generate_matrix(my_mat);
   pthread_mutex_lock(&mutex);
   Multiply_matrix(product_mat, my_mat);
   pthread_mutex_unlock(&mutex);
   Free_matrix(&my_mat);
   return NULL;
}  /* Thread_work */
```

使用pthreads发送消息的第一种尝试

```
/* messages has type char**. It's allocated in main. */
/* Each entry is set to NULL in main.                */
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   long source = (my_rank + thread_count - 1) % thread_count;
   char* my_msg = malloc(MSG_MAX*sizeof(char));

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;

   if (messages[my_rank] != NULL)
      printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
   else
      printf("Thread %ld > No message from %ld\n", my_rank, source);

   return NULL;
}  /* Send_msg */
```

各种信号量函数的语法

信号量不是pthread的一部分，需要在使用信号量的程序开头加上头文件.

```
#include <semaphore.h>

int sem_init(
      sem_t*     semaphore_p   /* out */,
      int        shared        /* in  */,
      unsigned   initial_val   /* in  */);

int sem_destroy(sem_t*   semaphore_p   /* in/out */);
int sem_post(sem_t*      semaphore_p   /* in/out */);
int sem_wait(sem_t*      semaphore_p   /* in/out */);
```

路障和条件变量

路障

- 通过确保所有线程在程序中处于同一位置来同步线程，这个同步点又称为路障。
- 只有所有线程都到达这一路障，线程才能继续运行下去，否则会阻塞在路障处。

使用路障来计时最慢的线程

```
/* Shared */
double elapsed_time;
. . .
/* Private */
double my_start, my_finish, my_elapsed;
. . .
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
. . .
Store current time in my_finish;
my_elapsed = my_finish - my_start;

elapsed = Maximum of my_elapsed values;
```

使用路障进行程序调试

```
point in program we want to reach;
barrier;
if (my_rank == 0) {
    printf("All threads reached this point\n");
    fflush(stdout);
}
```

忙等待和互斥量

- 使用忙等待和互斥量实现路障比较直观。
- 我们使用一个互斥量保护共享计数器。
- 当计数器的值表明每个线程都已经进入临界区，所有线程就可以离开忙等待的状态。

忙等待和互斥量

```
/* Shared and initialized by the main thread */
int counter;  /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&barrier_mutex);
    counter++;
    pthread_mutex_unlock(&barrier_mutex);
    while (counter < thread_count);
    . . .
}
```

每一个路障实例需要一个计数器变量，否则问题可能发生。

用信号量实现路障

```
/* Shared variables */
int counter;          /* Initialize to 0 */
sem_t count_sem;      /* Initialize to 1 */
sem_t barrier_sem;    /* Initialize to 0 */
. . .
void* Thread_work(...) {
   . . .
   /* Barrier */
   sem_wait(&count_sem);
   if (counter == thread_count-1) {
      counter = 0;
      sem_post(&count_sem);
      for (j = 0; j < thread_count-1; j++)
         sem_post(&barrier_sem);
   } else {
      counter++;
      sem_post(&count_sem);
      sem_wait(&barrier_sem);
   }
   . . .
```

条件变量

- 条件变量是一个数据对象，允许线程在某个特定条件或事件前都处于挂起状态.
- 当事件或条件发生时，另一个线程可以通过信号量来唤醒挂起的线程
- 条件变量总是与互斥量相关联.

条件变量

```
lock mutex;
if condition has occurred
   signal thread(s);
else {
   unlock the mutex and block;
   /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
```

用条件变量实现路障

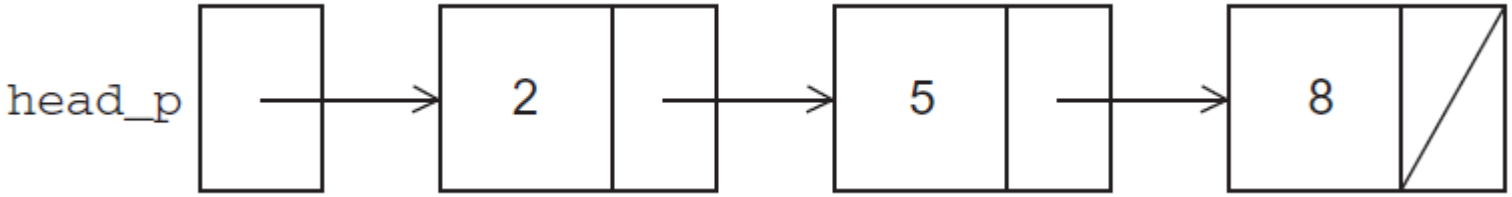
```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if (counter == thread_count) {
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    } else {
        while (pthread_cond_wait(&cond_var, &mutex) != 0);
    }
    pthread_mutex_unlock(&mutex);
    . . .
}
```

读写锁

对大型共享数据结构的控制访问

- 让我们看一个例子.假设共享的数据结构是一个存储int类型数据的链表，对链表的操作是Member、Insert 和 Delete。

链表

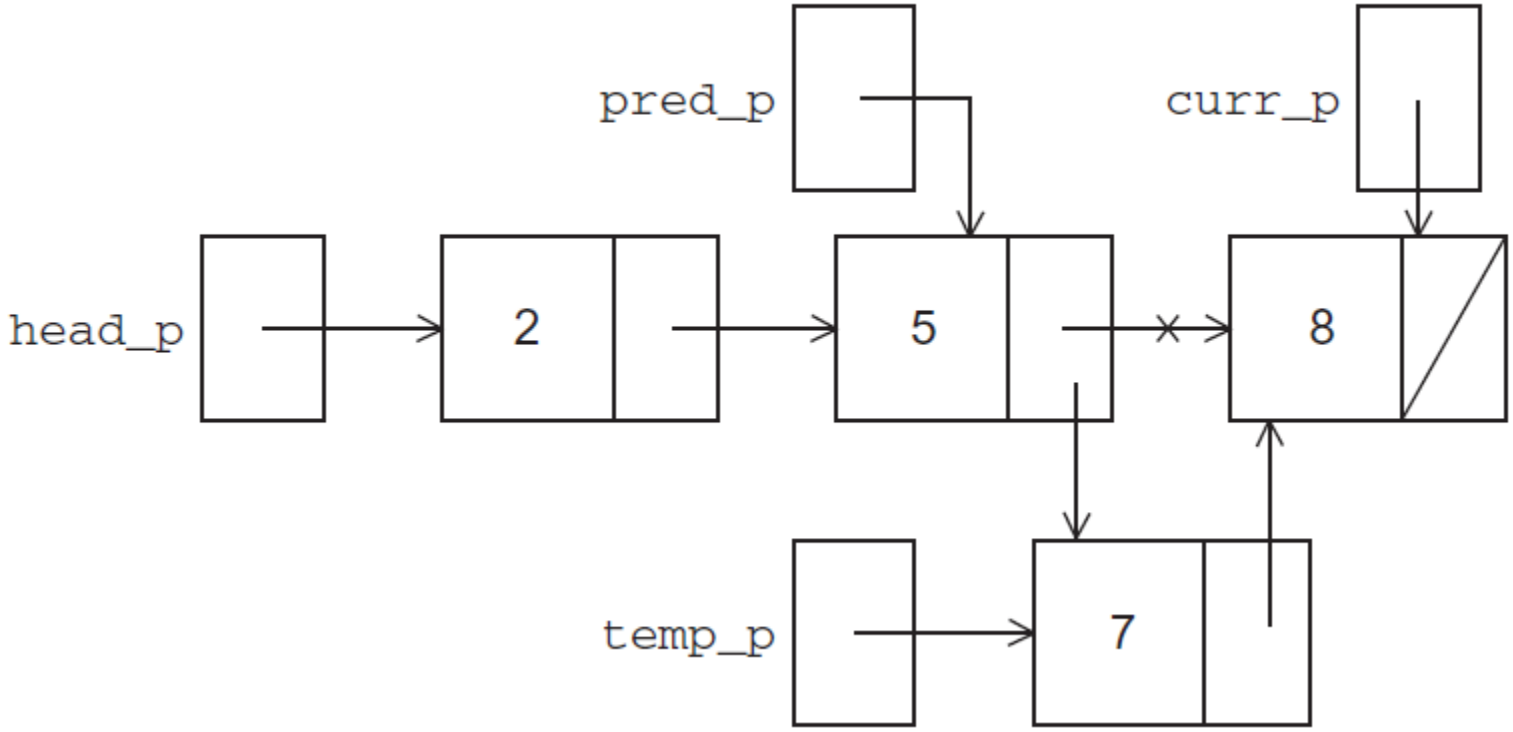



```
struct list_node_s {
    int data;
    struct list_node_s* next;
}
```

链表的Member函数

```
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 */
```

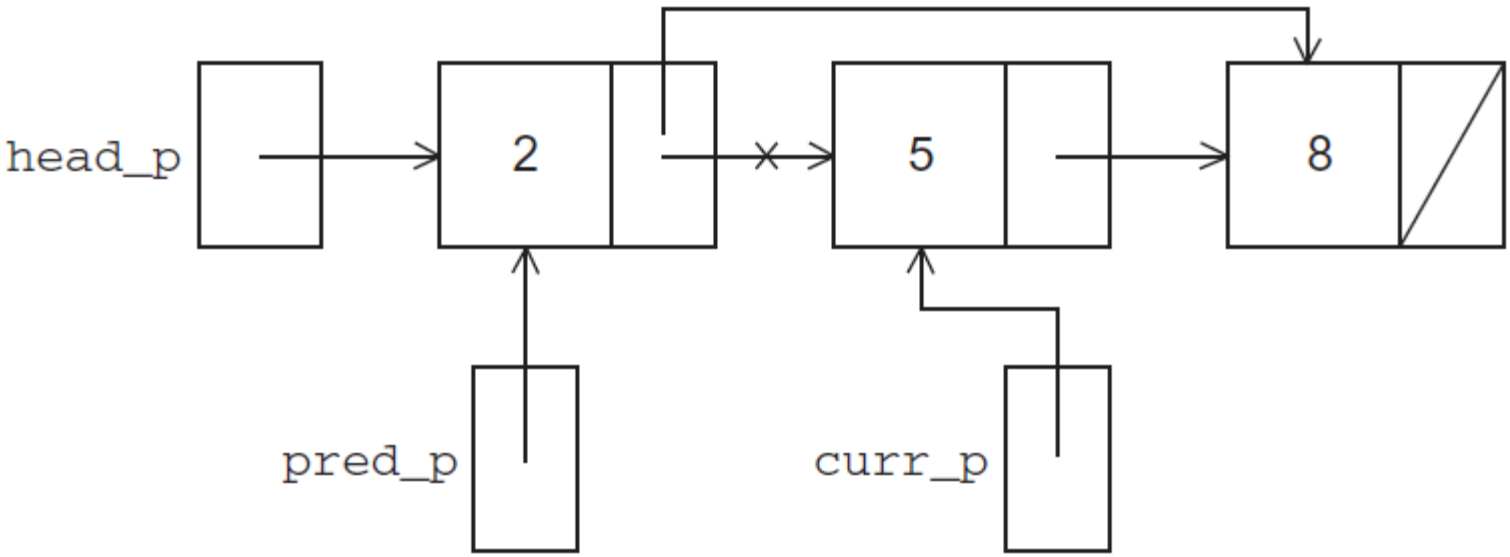
在链表中插入一个新结点



在链表中插入一个新结点

```
int Insert(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   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_pp = temp_p;
      else
         pred_p->next = temp_p;
      return 1;
   } else { /* Value already in list */
      return 0;
   }
}  /* Insert */
```

从链表中删除一个结点



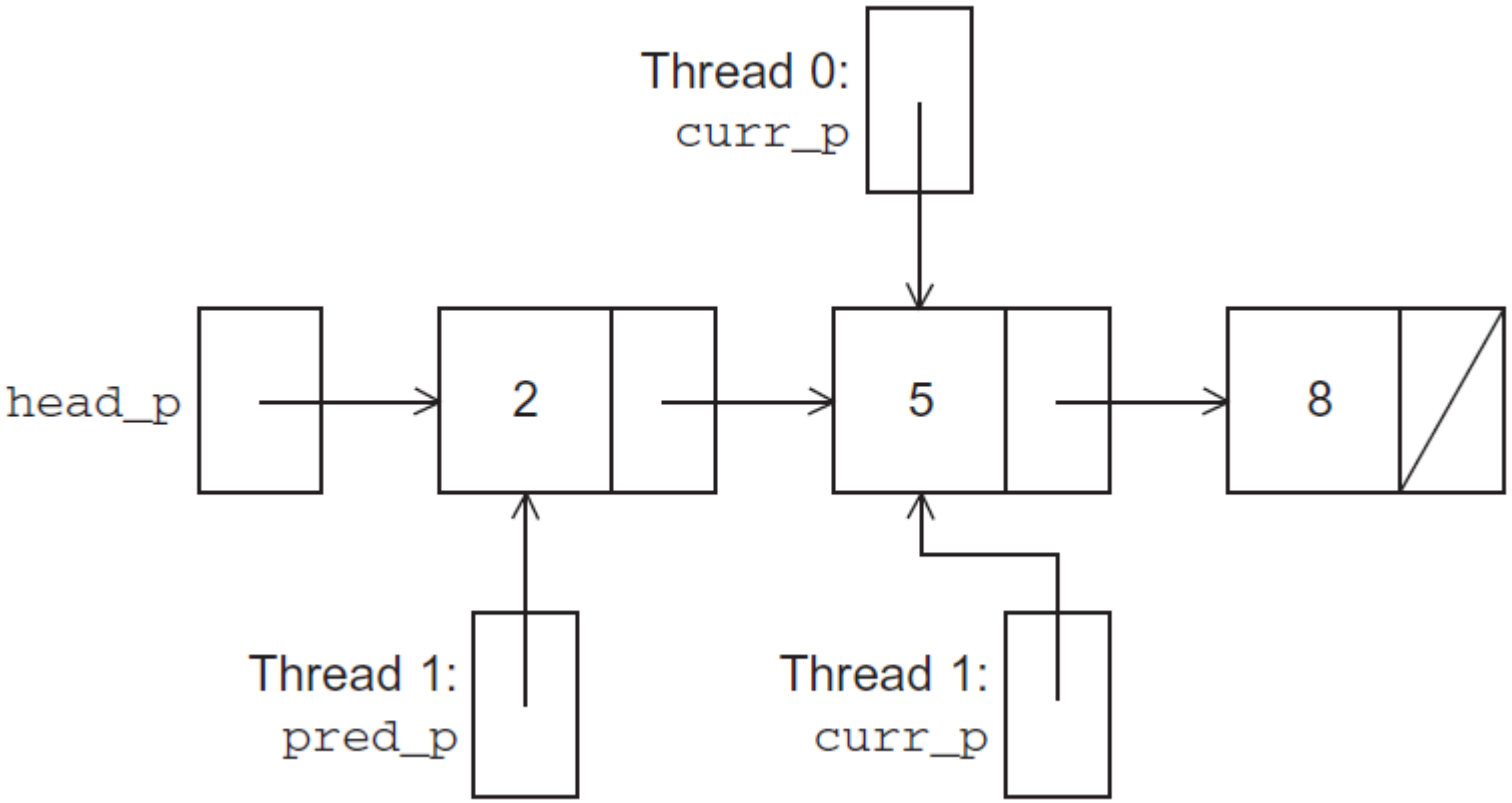
从链表中删除一个结点

```
int Delete(int value, struct list_node_s** head_pp) {
   struct list_node_s* curr_p = *head_pp;
   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_pp = 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 */
```

多线程链表

- 让我们尝试在Pthreads程序中使用这些函数。
- 为了对链表共享访问，我们可以将head_p定义为一个全局变量。
- 这将简化Member、Insert 和 Delete的函数头，因为我们不需要传入head_p或指向head_p的指针：我们只需要传递要插入的值。

两个线程同时访问链表



解决方案 #1

- 一个显而易见的解决方案是在线程试图访问链表前先加锁。
- 调用这三个函数的需要用互斥量保护.

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

代替 Member(value) 的调用.

问题来了

- 我们必须串行访问链表
- 如果我们对链表的操作绝大多数是调用Member，我们将失去开发并行性的机会。
- 另一方面，如果对链表的操作大多数都是对Insert 和 Delete 的调用，那么这可能是最好的解决方案，因为大多数的操作都需要串行执行，而且这个解决方案很容易实现。

解决方案 #2

- 我们可以尝试锁定单个结点，而不是锁定整个链表。
- “细粒度”锁。

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

问题来了

- 这比原来的Member函数实现更复杂。
- 它还比原来的实现慢，因为每次访问一个结点时，都必须对结点加锁和解锁。
- 向每个结点增加一个互斥量域，这必然增加了整个链表对存储量的需求。

每个链表结点拥有一个互斥量的方法实现Member函数 (1)

```
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;
   }
```

每个链表结点拥有一个互斥量的方法实现Member函数 (2)

```
    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 */
```

Pthreads读写锁

- 前面介绍的方法都不让正在执行Member函数的线程还可以同时访问链表的任意结点。
- 第一种解决方案任何时候只允许一个线程访问整个链表。
- 第二种解决方案任何时候只允许一个线程访问任一给定结点。

Pthreads读写锁

- 除了提供两个锁函数外，读写锁有点像互斥锁.
- 第一个为读操作对读写锁进行加锁，而第二个为写操作对读写锁进行加锁，

Pthreads读写锁

- 因此，多个线程可以通过调用读锁函数同时获得锁，而只有一个线程可以通过调用写锁函数获得锁。

- 因此，如果任何线程拥有读锁，那么任何请求写锁的线程将阻塞在写锁函数的调用上。

Pthreads读写锁

- 如果任何线程拥有了写锁，那么任何想要获得读或写锁的线程都将阻塞在其对应的锁函数上。

保护链表函数

```
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);
```

不同实现方案的性能

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

100,000 ops/thread

99.9% Member

0.05% Insert

0.05% Delete

不同实现方案的性能

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

100,000 ops/thread

80% Member

10% Insert

10% Delete

缓存、缓存一致性和伪共享

- 回想一下，芯片设计者在处理器中添加了一些相对快速的内存块，称为缓存（cache memory）。
- 缓存内存的使用对共享内存有很大的影响。
- 当一个核试图更新一个不在缓存中的变量时，就会发生写缺失，处理器必须访问主内存。

Pthreads矩阵-向量乘

```
void *Pth_mat_vect(void* rank) {
    long my_rank = (long) rank;
    int i, j;
    int local_m = m/thread_count;
    int my_first_row = my_rank*local_m;
    int my_last_row = (my_rank+1)*local_m - 1;

    for (i = my_first_row; i <= my_last_row; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i][j]*x[j];
    }

    return NULL;
} /* Pth_mat_vect */
```

矩阵-向量乘法的运行时间和效率

Threads	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

(时间以秒为单位)

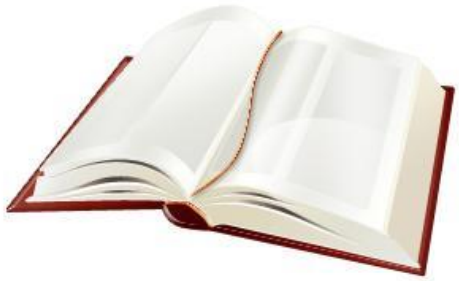
线程安全性

线程安全性

- 如果一个代码块可以被多个线程同时执行而不会引起问题，那么它就是线程安全的。

Example

- 假设我们想要使用多个线程来对一个文件进行“分词”。
- 假设文件由普通的英语文本构成，要分析出的是空格、tab或换行符分割的连续的字符序列。



简单方法

- 将输入文件划分为行，并以循环的方式将这些行分配给线程。
- 第一行给线程0，第二行给线程1，…，第t行给线程t，第t +1行给线程0，等等

简单方法

- 我们可以使用信号量将访问输入行串行化.
- 在一个线程读取了一行输入后，它就能够对这一行使用strtok函数进行分词。

strtok 函数

- 第一次调用它时，string参数应该是要被分词的文本。
  - 我们的例子中就是一行输入.
- 对于后续调用，第一个参数应该是NULL。

```
char* strtok(
        char*       string      /* in/out */,
        const char* separators  /* in     */);
```

strtok 函数

- 其思想是，在第一次调用时，strtok缓存一个指向string的指针，对于后续的调用，它返回从缓存副本中分割出的连续的词。

多线程分词器的第一个版本(1)

```
void *Tokenize(void* rank) {
   long my_rank = (long) rank;
   int count;
   int next = (my_rank + 1) % thread_count;
   char *fg_rv;
   char my_line[MAX];
   char *my_string;

   sem_wait(&sems[my_rank]);
   fg_rv = fgets(my_line, MAX, stdin);
   sem_post(&sems[next]);
   while (fg_rv != NULL) {
      printf("Thread %ld > my line = %s", my_rank, my_line);
```

多线程分词器的第一个版本(2)

```
        count = 0;
        my_string = strtok(my_line, " \t\n");
        while ( my_string != NULL ) {
            count++;
            printf("Thread %ld > string %d = %s\n", my_rank, count,
                  my_string);
            my_string = strtok(NULL, " \t\n");
        }

        sem_wait(&sems[my_rank]);
        fg_rv = fgets(my_line, MAX, stdin);
        sem_post(&sems[next]);
    }

    return NULL;
}  /* Tokenize */
```

用一个线程运行

- 它能正确地对输入流进行分词

Pease porridge hot.

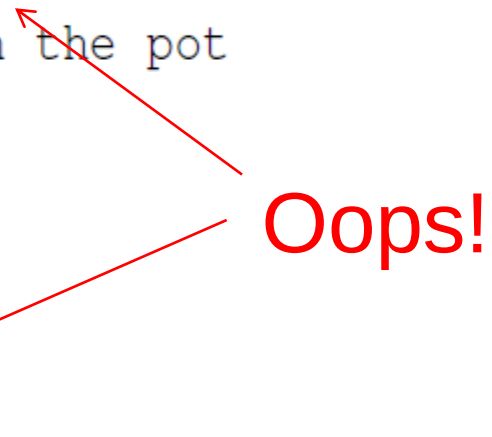
Pease porridge cold.

Pease porridge in the pot

Nine days old.

使用两个线程运行

```
Thread 0 > my line = Pease porridge hot.
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = hot.
Thread 1 > my line = Pease porridge cold.
Thread 0 > my line = Pease porridge in the pot
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = in
Thread 0 > string 4 = the
Thread 0 > string 5 = pot
Thread 1 > string 1 = Pease
Thread 1 > my line = Nine days old.
Thread 1 > string 1 = Nine
Thread 1 > string 2 = days
Thread 1 > string 3 = old.
```



怎么回事?

- Strtok通过声明一个Storge类的静态变量来实现对输入行的缓存.
- 这将导致从本次调用到下次调用之间，存储在该变量中的值会一直被保留
- 不幸的是，这个缓存中的字符串是共享的，而不是私有的。

怎么回事?

- 因此，线程0调用strtok对输入的第3行进行缓存，覆盖了原来线程1调用strtok对输入的第2行的缓存。
- 因此，strtok函数不是线程安全的。如果多个线程同时调用它，输出可能不正确。

其他不安全的C库函数

- 遗憾的是，对于C标准库函数来说，线程不安全是常见的。
- stdlib.h中的随机数生成器函数random。
- time.h中的时间转换函数localtime。

“可重入”函数 (线程安全)

- 在某些情况下，C标准指定了一个替代的、线程安全的函数版本。

```
char* strtok_r(
        char*          string      /* in/out */,
        const char*    separators, /* in     */
        char**         saveptr_p   /* in/out */);
```

小结 (1)

- 共享内存编程中的线程类似于分布式内存编程中的进程.
- 然而，线程通常比进程更轻量级.
- 在Pthreads程序中，所有线程都可以访问全局变量，而局部变量对于运行程序的线程来说是私有变量。

小结(2)

- 当多个线程试图访问一个共享资源时，如共享变量或共享文件，并且其中至少有一个访问是更新操作，这样的访问可能会导致错误，导致结果的不确定性，我们称这种现象为竞争条件。

小结(3)

- 临界区是一个代码块，在这个代码块中，任意时刻只有一个线程能够更新共享资源。
- 因此，临界区中的代码执行应该应该作为串行代码执行。

小结(4)

- 忙等待可使用一个标志变量和一个空while循环实现，来避免对临界区的访问冲突。
- 它非常浪费CPU周期。
- 如果打开编译器优化，它可能是不可靠的。

小结(5)

- 互斥量可以被看做是临界区的一把锁，用来避免临界区的访问冲突，它可以保证对临界区里的互斥访问

小结(6)

- 信号量是避免临界区访问冲突的第3种方法.
- 它是一个unsigned int类型，具有两个操作:sem_wait和sem_post.
- 信号量比互斥量功能更强大，因为它们可以被初始化为任何非负值。

小结(7)

- 路障是程序中的一个结点，线程必须阻塞，直到所有线程都到达这个结点为止。
- 当多个线程同时安全地读一个数据结构时，可以使用读写锁；如果线程需要修改或者写数据时，在修改期间，只有一个线程能够访问该数据结构。

小结(8)

- 某些C函数通过申明static变量，从而在两次调用之间缓存数据。当多个线程调用该函数时，可能会导致错误。
- 这种类型的函数不是线程安全的.