并行计算

（Parallel Computing）

共享内存编程 - Pthreads

学习内容：

- 进程、线程、Pthreads
- Hello，World
- Pthreads中的矩阵 – 向量乘法
- 临界区（Critical Section）
- 忙 – 等（Busy-Waiting）

共享内存编程 - Pthreads

学习内容：

- 互斥量（Mutex）
- 生产者 – 消费者同步和信号量（Semaphore）
- 屏障（Barrier）和条件变量（Condition Variable）
- 读 – 写锁（Read-Write Lock）
- Cache，Cache一致性，伪共享（False Sharing）
- 线程安全性

8.屏障（Barrier）和条件变量（Condition Variable）

- 屏障（Barrier）：信号量

Can

counter,

count_sem,

barrier_sem

be reused ?

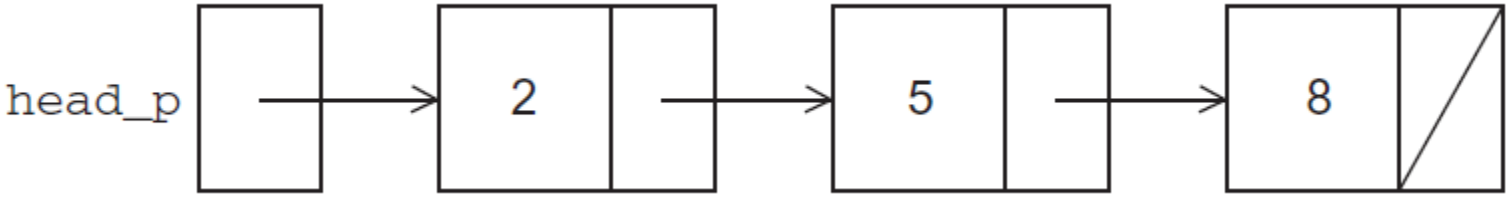
```
/* Shared variables */
int counter;          /* Initialize to 0 */
sem_t count_sem;      /* Initialize to 1 */
sem_t barrier_sem;    /* Initialize to 0 */
. . .
void* Thread_work(...) {
   . . .
   /* Barrier */
   sem_wait(&count_sem);
   if (counter == thread_count-1) {
      counter = 0;
    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

9.读 – 写锁（Read-Write Lock）

- 控制对大型共享数据结构的访问
  - 假设共享数据结构是一个已排序的int链表
  - 感兴趣的操作是 Member、Insert 和 Delete

9.读 – 写锁（Read-Write Lock）

- 控制对大型共享数据结构的访问




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

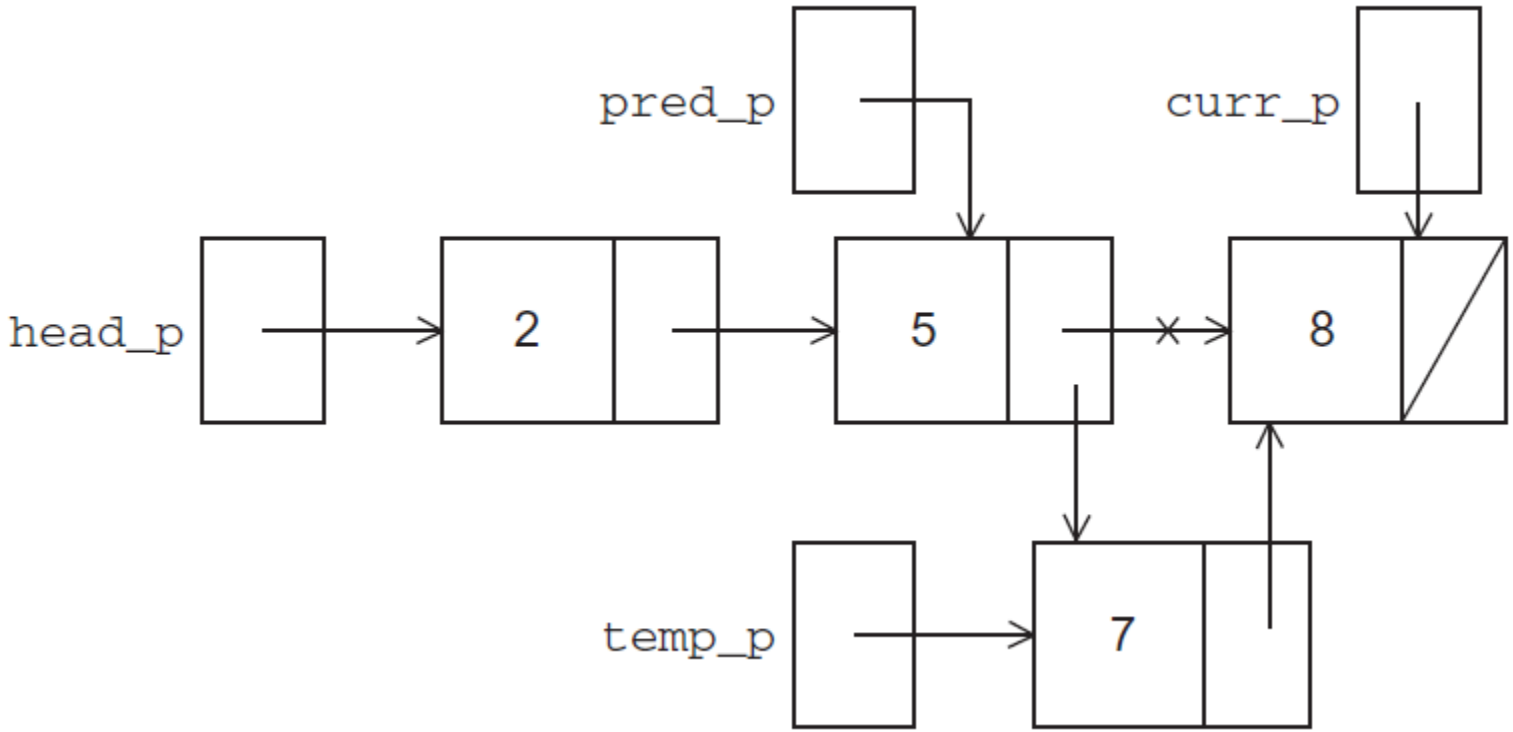
9.读 – 写锁（Read-Write Lock）

- 确认是否为链表中的元素

```
1  int  Member(int value, struct list_node_s* head_p) {
2     struct list_node_s* curr_p = head_p;
3
4     while (curr_p != NULL && curr_p->data < value)
5        curr_p = curr_p->next;
6
7     if (curr_p == NULL || curr_p->data > value) {
8        return 0;
9     } else {
10       return 1;
11    }
12 }  /* Member */
```

9.读 – 写锁（Read-Write Lock）

- 在链表中插入一个新的节点

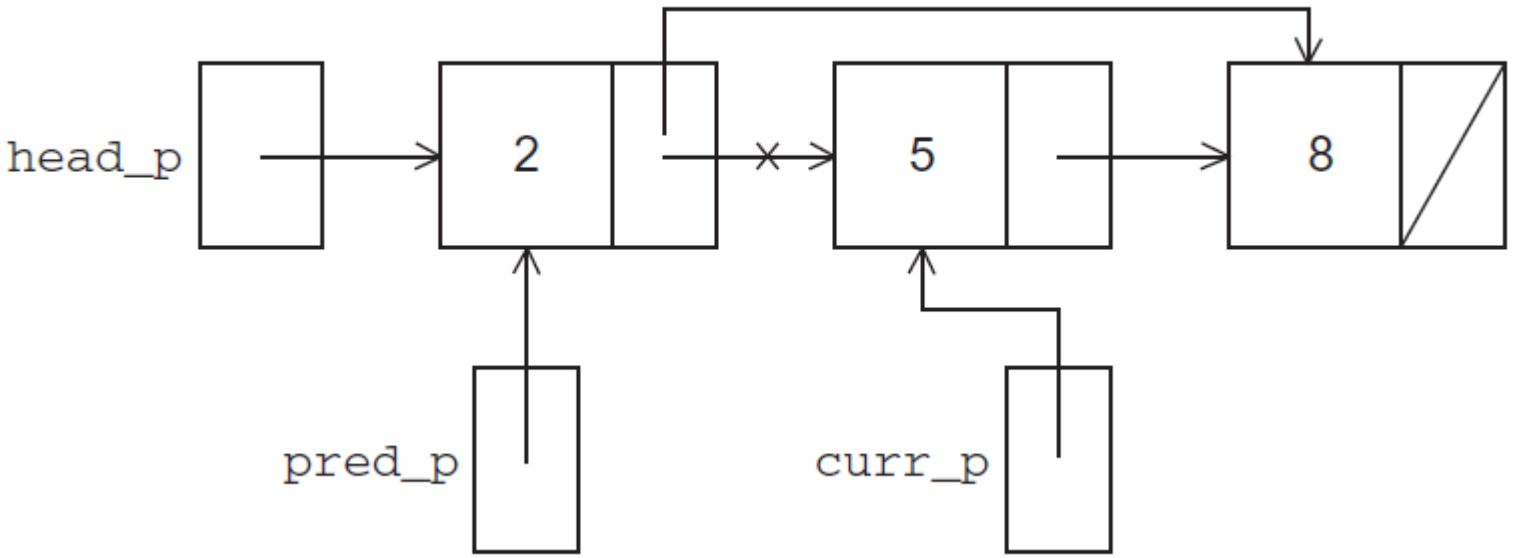


9.读 – 写锁（Read-Write Lock）

```
1  int Insert(int value, struct list_node_s** head_p) {
2     struct list_node_s* curr_p = *head_p;
3     struct list_node_s* pred_p = NULL;
4     struct list_node_s* temp_p;
5
6     while (curr_p != NULL && curr_p->data < value) {
7        pred_p = curr_p;
8        curr_p = curr_p->next;
9     }
10
11    if (curr_p == NULL || curr_p->data > value) {
12       temp_p = malloc(sizeof(struct list_node_s));
13       temp_p->data = value;
14       temp_p->next = curr_p;
15       if (pred_p == NULL)  /* New first node */
16          *head_p = temp_p;
17       else
18          pred_p->next = temp_p;
19       return 1;
20    } else { /* Value already in list */
21       return 0;
22    }
23 }  /* Insert */
```

9.读 – 写锁（Read-Write Lock）

- 在链表中删除一个节点



9.读 – 写锁（Read-Write Lock）

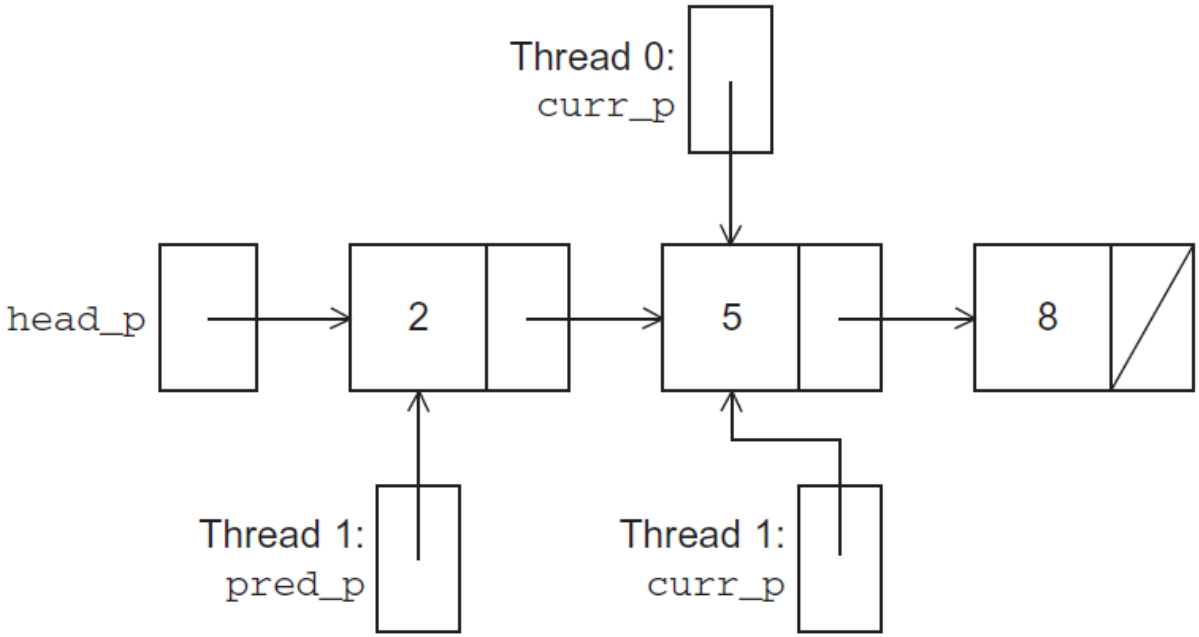
```
1  int Delete(int value, struct list_node_s** head_p) {
2     struct list_node_s* curr_p = *head_p;
3     struct list_node_s* pred_p = NULL;
4
5     while (curr_p != NULL && curr_p->data < value) {
6        pred_p = curr_p;
7        curr_p = curr_p->next;
8     }
9
10    if (curr_p != NULL && curr_p->data == value) {
11       if (pred_p == NULL) { /* Deleting first node in list */
12          *head_p = curr_p->next;
13          free(curr_p);
14       } else {
15          pred_p->next = curr_p->next;
16          free(curr_p);
17       }
18       return 1;
19    } else {  /* Value isn't in list */
20       return 0;
21    }
22 }  /* Delete */
```

9.读 – 写锁（Read-Write Lock）

- 控制对大型共享数据结构的访问
  - 如果多个进程共享访问链表，如：head_p 为全局变量
  - 当多个进程同时访问链表，进行操作，结果如何？

9.读 – 写锁（Read-Write Lock）

- 控制对大型共享数据结构的访问
  - 线程 0 执行 Member(5)
  - 同时，线程 1 执行 Delete(5)
  - 线程 0 执行 Member(8)



9.读 – 写锁（Read-Write Lock）

- 控制对大型共享数据结构的访问
  - 解决方法 1：当进程试图访问链表时，对其加锁

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

9.读 – 写锁（Read-Write Lock）

- 控制对大型共享数据结构的访问
  - 解决方法 1：当进程试图访问链表时，对其加锁
    - 串行化链表的访问
    - 如果大多数操作为 Member，则没有利用到并行化
    - 如果大多数操作为 Insert 和 Delete，则为易实现的方案

9.读 – 写锁（Read-Write Lock）

- 控制对大型共享数据结构的访问
  - 解决方法 2：为链表中的单个节点加锁（“细粒度”方法）

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

9.读 – 写锁（Read-Write Lock）

- 控制对大型共享数据结构的访问

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

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;

   /* If list is not empty, acquire the mutex
    * associated with first node */
   if (temp_p != NULL)
      pthread_mutex_lock(temp_p->mutex);

   /* Don't need head_p mutex anymore */
   pthread_mutex_unlock(&head_p_mutex);
```

9.读 – 写锁（Read-Write Lock）

```
while (temp_p != NULL && temp_p->data < value) {
   if (temp_p->next != NULL)
      pthread_mutex_lock(&(temp_p->next->mutex));

   /* Advance to next element */
   old_temp_p = temp_p;
   temp_p = temp_p->next;

   /* Now unlock previous element's mutex */
   pthread_mutex_unlock(&(old_temp_p->mutex));
}

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

9.读 – 写锁（Read-Write Lock）

- 控制对大型共享数据结构的访问
  - 解决方法 2：为链表中的单个节点加锁（“细粒度”方法）
    - 比原始的操作函数复杂
    - 速度慢，每次一个节点被访问，将调用 lock 和 unlock
    - 增加了链表的存储空间

9.读 – 写锁（Read-Write Lock）

- 控制对大型共享数据结构的访问
  - Pthreads 读写锁
    - 类似于 mutex，但提供了两个 lock 函数
    - 第一个 lock 函数用来读，第二个 lock 函数用来写
    - 多个进程可以同时获得读操作的锁，而只有一个进程可以获得写操作的锁
    - 如果一个进程获得了读操作的锁，其他想获得写操作锁的进程将被阻塞
    - 如果一个进程获得了写操作的锁，其他想获得读或写操作锁的进程将被阻塞

9.读 – 写锁（Read-Write Lock）

- 控制对大型共享数据结构的访问
  - Pthreads 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);
```

9.读 – 写锁（Read-Write Lock）

- 控制对大型共享数据结构的访问

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

1000 keys

100,000 ops/thread

99.9% Member

0.05% Insert

0.05% Delete

9.读 – 写锁（Read-Write Lock）

- 控制对大型共享数据结构的访问

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

1000 keys

100,000 ops/thread

80% Member

10% Insert

10% Delete

10. Cache，Cache一致性，伪共享（False Sharing）

- Cache 的设计遵循“时间-空间局部性”原则
- 一次读取的内存块称为“cache line”或“cache block”
- Cache一致性会对共享内存系统的性能产生巨大影响
- write-miss：当core试图更新不在缓存中的变量
- read-miss：当core试图读取不在缓存中的变量

10. Cache，Cache一致性，伪共享（False Sharing）

```
1   void *Pth_mat_vect(void* rank) {
2      long my_rank = (long) rank;
3      int i, j;
4      int local_m = m/thread_count;
5      int my_first_row = my_rank*local_m;
6      int my_last_row = (my_rank+1)*local_m - 1;
7
8      for (i = my_first_row; i <= my_last_row; i++) {
9         y[i] = 0.0;
10        for (j = 0; j < n; j++)
11            y[i] += A[i][j]*x[j];
12     }
13
14     return NULL;
15  }  /* Pth_mat_vect */
```

10. Cache，Cache一致性，伪共享（False Sharing）

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

(times are in seconds)

$$E = \frac{S}{t} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{t} = \frac{T_{\text{serial}}}{t \times T_{\text{parallel}}}$$

10. Cache，Cache一致性，伪共享（False Sharing）

- 缓存一致性是在“cache line”级别执行的，也就是说，每次写入cache line 中的任何值时，则整行都将失效，而不仅仅是写入的值
- 例如：一条缓存线为64字节，y为double（8字节），则可存储8个double
- 两个双核处理器，每个处理器有 cache，假设线程 0 和 1 被分配到一个处理器上，线程 2 和 3 被分配到另一个处理器上

10. Cache，Cache一致性，伪共享（False Sharing）

- 对于 8 * 8000000 的情况，所有的 y 存放在一条缓存线上，每次对其元素的写操作都会导致另一个处理器上的缓存线失效

```
y[i] += A[i][j]*x[j];
```

- 伪共享：具有不同缓存的两个线程访问属于同一缓存线的不同变量。线程不共享任何内容，但线程在内存访问方面的行为与共享变量时的行为相同

10. Cache，Cache一致性，伪共享（False Sharing）

- 对于 8000 * 8000 的情况：假设线程2被分配到一个处理器，线程3被分配到另一个处理器
- 线程2负责计算：　y[4000], y[4001], . . . , y[5999]
- 线程3负责计算：　y[6000], y[6001], . . . , y[7999]
- 只有在交界处可能发生“伪共享”

y[5996], y[5997], y[5998], y[5999]　　y[6000], y[6001], y[6002], y[6003]

11.线程安全性

- 如果一段代码可以由多个线程同时执行而不会引起问题，那么它是线程安全的（thread-safe）
- 例如：用多个线程来“标记化”（tokenize）一个文件
  - 标记（tokens）是连续的字符序列，用空格、制表符或换行符分隔

11.线程安全性

- 将输入文件分成行文本，并以循环方式将这些行分配给线程
  - 第一行分给线程 0，第二行分给线程 1，第 t 行分给线程 t - 1，第 t + 1 行分给线程 0，…
  - 用信号量控制对输入行的访问
  - 一个线程读取一行后，用 strtok 函数来对行进行标记化

11.线程安全性

- strtok 函数

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

  - 第一次调用时，参数 string 应指向被标记化的文本，即：一行输入
  - 后续调用，参数 string 为 NULL
  - 在第一次调用中，strtok 缓存一个指向字符串的指针，对于随后的调用，它返回从缓存副本中获取的标记

```
1  void* Tokenize(void* rank) {
2     long my_rank = (long) rank;
3     int count;
4     int next = (my_rank + 1) % thread_count;
5     char *fg_rv;
6     char my_line[MAX];
7     char *my_string;
8
9     sem_wait(&sems[my_rank]);
10    fg_rv = fgets(my_line, MAX, stdin);
11    sem_post(&sems[next]);
12    while (fg_rv != NULL) {
13       printf("Thread %ld > my line = %s", my_rank, my_line);
14
15       count = 0;
16       my_string = strtok(my_line, " \t\n");
17       while ( my_string != NULL ) {
18          count++;
19          printf("Thread %ld > string %d = %s\n", my_rank, count,
20                my_string);
21          my_string = strtok(NULL, " \t\n");
22       }
23
24       sem_wait(&sems[my_rank]);
25       fg_rv = fgets(my_line, MAX, stdin);
26       sem_post(&sems[next]);
27    }
28
29    return NULL;
30 }  /* Tokenize */
```

11.线程安全性

- 运行一个线程：正确标记

Pease porridge hot.

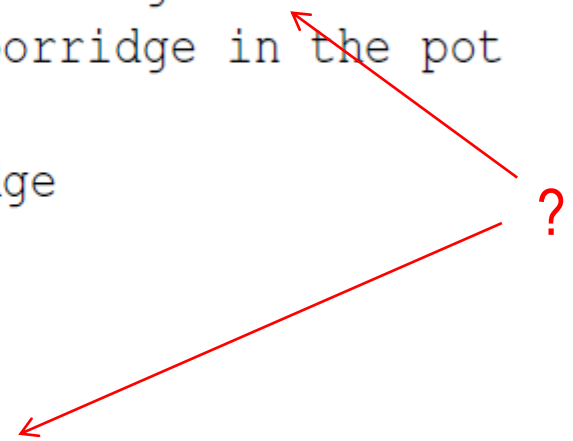
Pease porridge cold.

Pease porridge in the pot

Nine days old.

11.线程安全性：运行两个线程

```
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.
```



11.线程安全性

- What happened?
  - strtok 通过声明静态变量来缓存输入行
  - 这将导致存储在该变量中的值在调用之间保持不变
  - 这个缓存字符串是共享的，而不是私有的

11.线程安全性

- What happened?
  - 线程 0 用输入的第三行调用 strtok，覆盖了线程 1 用第二行调用的内容
  - 因此 strtok 函数不是线程安全的。如果多个线程同时调用它，则输出可能不正确

11.线程安全性

- 其他不安全的 C 库函数
  - C库函数不能保证线程安全的情况并不少见
  - stdlib.h 中的随机数发生器 random
  - time.h 中的时间转换函数 localtime

11.线程安全性

- 可重入（re-entrant）（线程安全）函数
  - C 标准提供了替代的线程安全的版本函数

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

小结

- 共享内存程序中的线程类似于分布式内存程序中的进程
- 然而，一个线程通常比一个进程更轻
- 在 Pthreads 程序中，所有线程都可以访问全局变量，而局部变量通常是运行该函数的线程的私有变量

小结

- 当多个线程试图访问共享资源（如共享变量或共享文件），并且至少有一个访问是更新操作，将导致不确定性且可能导致错误
- 临界区（critical section）是更新共享资源的代码块，共享资源一次只能由一个线程更新，即临界区中的代码以串行方式执行

小结

- 忙-等（Busy-waiting）可以用标志变量和 while 循环来避免临界区的访问冲突
- 浪费 CPU 时间
- 编译器有可能对其进行优化

小结

- 互斥量（mutex）也可以用来避免对临界区的访问冲突
- 可以将其看作是在临界区上的“锁”

小结

- 信号量（semaphore）是避免对临界区访问冲突的第三种方法
- 为 unsigned int，sem_wait 和 sem_post 两个操作
- 信号量比互斥量更强大，因为它们可以初始化为任何非负值
- 信号量可以用作“生产者-消费者”的同步

小结

- 屏障（barrier）是程序中的一个点，在这个点上线程阻塞，直到所有线程都到达它为止
- 读-写锁（read-write lock）：当多个线程可以安全地同时读取一个数据结构时，使用读写锁，但是如果一个线程需要修改或写入该数据结构，则只有该线程可以在修改期间访问该数据结构

小结

- 有些 C 函数通过声明变量为静态变量来缓存调用之间的数据，当多个线程调用该函数时会导致错误
- 这种类型的函数不是线程安全的

第三次作业

- 编写 Pthreads 程序实现梯形法则求面积
  - 使用共享变量对线程的计算结果进行累加
  - 使用 busy-waiting, mutexes 和 semaphores 实现对临界区域的互斥
  - 根据结果分析每种方法的优缺点