多线程，Pthread

于策

Outline

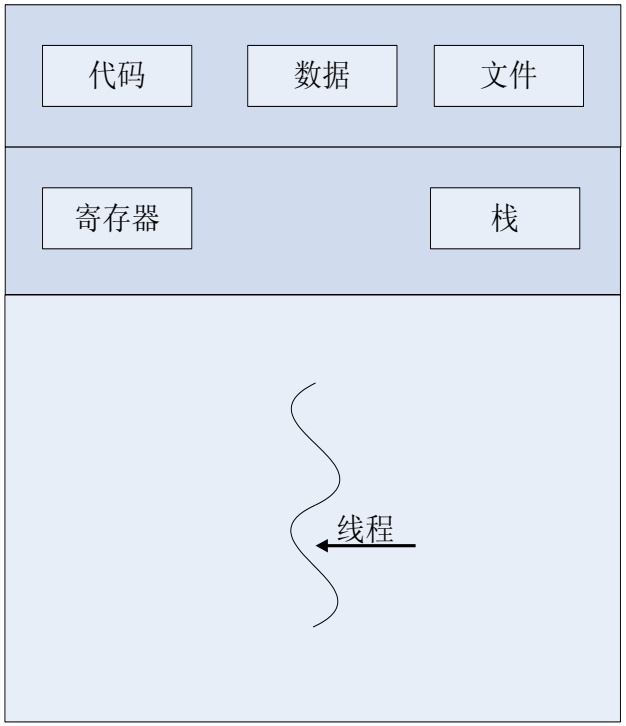
- 多线程基本概念
  - 线程与进程的区别
  - 线程的生命周期
  - 线程同步
- Pthread多线程
- 实例分析
  - 计算数组中“3”出现的次数

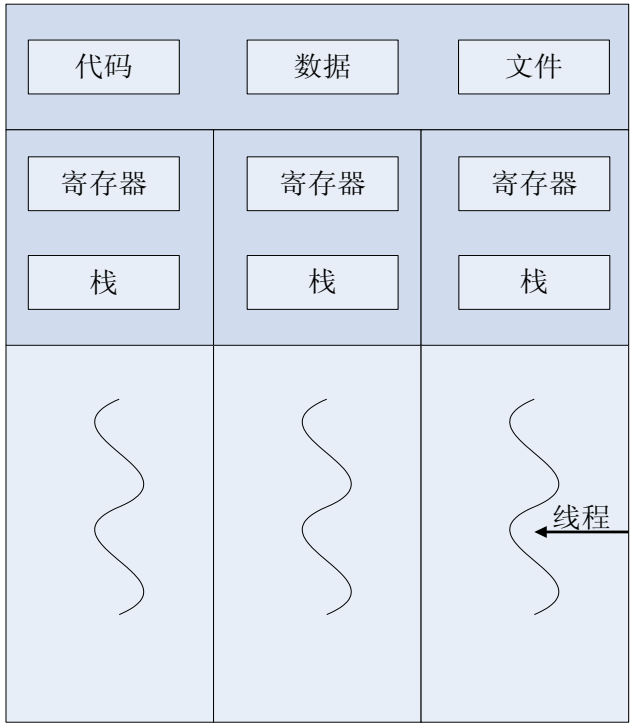
Outline

- 多线程基本概念
  - 线程与进程的区别
  - 线程的生命周期
  - 线程同步
- Pthread多线程
- 实例分析
  - 计算数组中“3”出现的次数

多线程概念

- 线程（thread）是进程上下文（context）中执行的代码序列，又被称为轻量级进程（light weight process）
- 在支持多线程的系统中，进程是资源分配的实体，而线程是被调度执行的基本单元。





线程与进程的区别

- 调度
- 并发性
- 拥有资源
- 系统开销

调度

- 在传统的操作系统中，CPU调度和分派的基本单位是进程。
- 在引入线程的操作系统中，则把线程作为CPU 调度和分派的基本单位，进程则作为资源拥有的基本单位，从而使传统进程的两个属性分开，线程便能轻装运行，这样可以显著地提高系统的并发性。
- 同一进程中线程的切换不会引起进程切换，从而避免了昂贵的系统调用。
  - 但是在由一个进程中的线程切换到另一进程中的线程时，依然会引起进程切换。

并发性

- 在引入线程的操作系统中，不仅进程之间可以并发执行，而且在一个进程中的多个线程之间也可以并发执行，因而使操作系统具有更好的并发性，从而能更有效地使用系统资源和提高系统的吞吐量。
  - 例如，在一个未引入线程的单CPU操作系统中，若仅设置一个文件服务进程，当它由于某种原因被封锁时，便没有其他的文件服务进程来提供服务。
- 在引入了线程的操作系统中，可以在一个文件服务进程中设置多个服务线程。
  - 当第一个线程等待时，文件服务进程中的第二个线程可以继续运行；当第二个线程封锁时，第三个线程可以继续执行，从而显著地提高了文件服务的质量以及系统的吞吐量。

拥有资源

- 进程
  - 不论是引入了线程的操作系统，还是传统的操作系统，进程都是拥有系统资源的一个独立单位，它可以拥有自己的资源。
- 线程
  - 线程自己不拥有系统资源（除部分必不可少的资源，如栈和寄存器），但它可以访问其隶属进程的资源。亦即一个进程的代码段、数据段以及系统资源（如已打开的文件、I/O设备等），可供同一进程的其他所有线程共享。

系统开销

- 进程
  - 创建或撤消进程时，系统都要为之分配或回收资源，如内存空间、I/O 设备等。
  - 在进行进程切换时，涉及到整个当前进程CPU 环境的保存环境的设置以及新被调度运行的进程的CPU 环境的设置。
- 线程
  - 切换只需保存和设置少量寄存器的内容，并不涉及存储器管理方面的操作。
  - 此外，由于同一进程中的多个线程具有相同的地址空间，致使它们之间的同步和通信的实现也变得比较容易。在有的系统中，线程的切换、同步和通信都无需操作系统内核的干预。

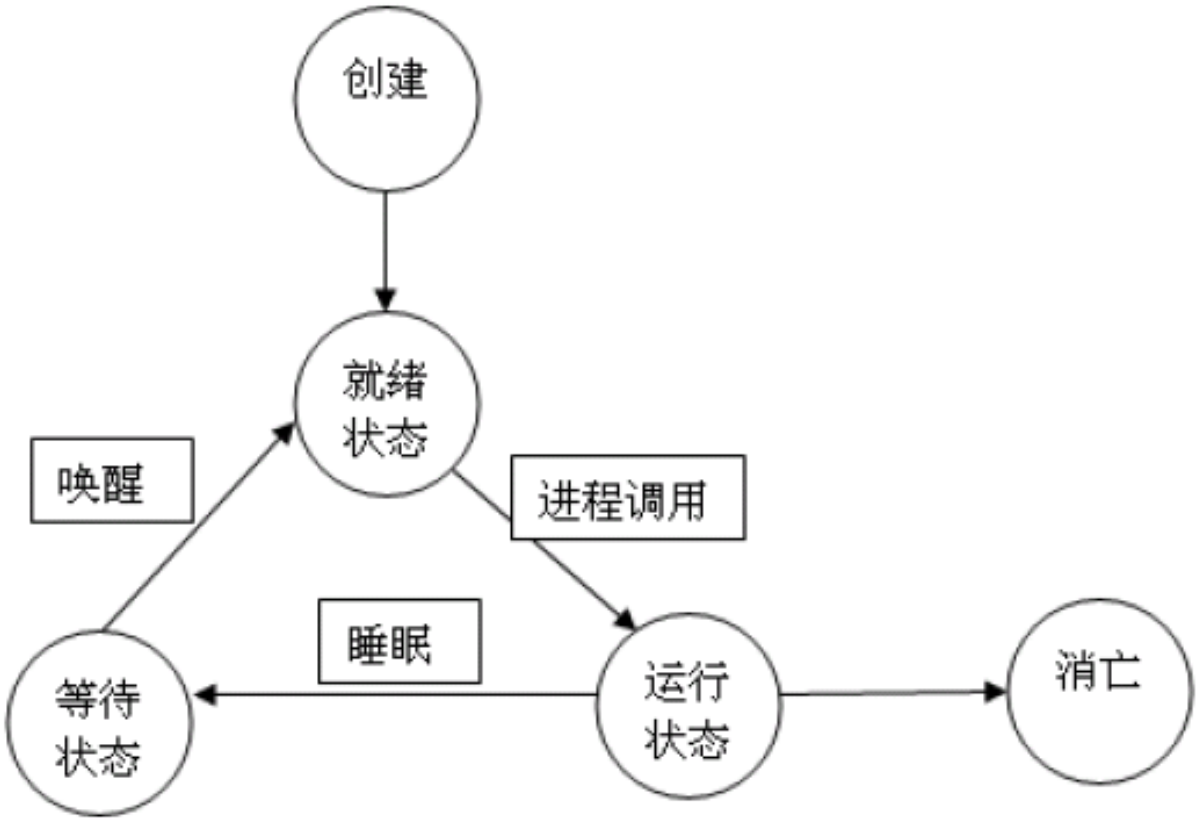
线程层次

- **用户级线程**在用户层通过线程库来实现。对它的创建、撤销和切换都不利用系统的调用。
- **核心级线程**由操作系统直接支持，即无论是在用户进程中的线程，还是系统进程中的线程，它们的创建、撤消和切换都由核心实现。
- **硬件线程**就是线程在硬件执行资源上的表现形式。
- 单个线程一般都包括上述三个层次的表现：用户级线程通过操作系统被作为核心级线程实现，再通过硬件相应的接口作为硬件线程来执行。

Outline

- **多线程基本概念**
  - 线程与进程的区别
  - **线程的生命周期**
  - 线程同步
- Pthread多线程
- 实例分析
  - 计算数组中“3”出现的次数

线程的生命周期



Outline

- **多线程基本概念**
  - 线程与进程的区别
  - 线程的生命周期
  - **线程同步**
- Pthread多线程
- 实例分析
  - 计算数组中“3”出现的次数

线程的同步

- 竞争条件
- 同步方法
  - 临界区
  - 信号量
  - 互斥锁
  - 条件变量

竞争条件

- 在有些操作系统中，多个线程间可能会共享一些彼此都能够读写的公用存储区。
  - 可以在内存中，也可以是一个共享文件。
- 当两个或多个进程试图在同一时刻访问共享内存，或读写某些共享数据，而最后的结果取决于线程执行的顺序(线程运行时序)，就称为竞争条件(Race Conditions)

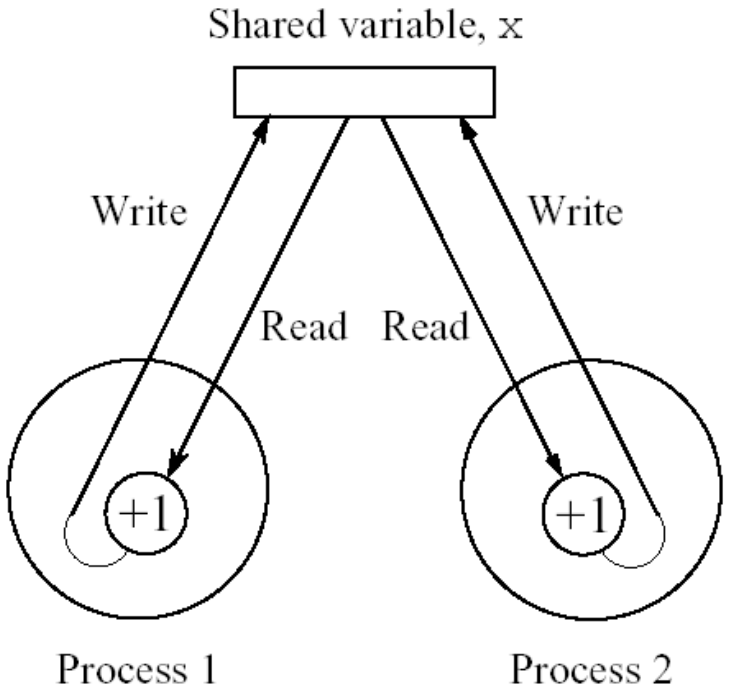
访问共享变量时产生的冲突

Instruction

```
x = x + 1;
```

Time

Process 1	Process 2
read x	read x
compute x + 1	compute x + 1
write to x	write to x



Bernstein 条件

$$I_1 \cap O_2 = \phi$$
$$I_2 \cap O_1 = \phi$$
$$O_1 \cap O_2 = \phi$$

I_i is the set of memory locations read by process P_i

O_j is the set of memory locations altered by process P_j

如果以上三个条件都得到了满足，则两个线程可以同步执行

临界区

- 临界区(critical section)是指包含有共享数据的一段代码，这些代码可能被多个线程执行。临界区的存在就是为了保证当有一个线程在临界区内执行的时候，不能有其他任何线程被允许在临界区执行。
- 设想有A,B两个线程执行同一段代码，则在任意时刻至多只能有一个线程在执行临界区内的代码。即，如果A线程正在临界区执行，B线程则只能在进入区等待。只有当A 线程执行完临界区的代码并退出临界区，原先处于等待状态的B线程才能继续向下执行并进入临界区。

信号量

- 信号量是E. W. Dijkstra 在1965 年提出的一种方法，可以用一个整数变量sem 来表示，对信号量有两个基本的原子操作：P（wait, 减量操作）和V（signal,增量操作）

对信号量的解释

- 以一个停车场的运作为例。
  - 假设停车场只有三个车位，一开始三个车位都是空的。这时如果同时来了五辆车，看门人允许其中三辆直接进入，然后放下车拦，剩下的车则必须在入口等待，此后来的车也都不得不在入口处等待。这时，有一辆车离开停车场，看门人得知后，打开车拦，放入外面的一辆进去，如果又离开两辆，则又可以放入两辆，如此往复。
- 在这个停车场系统中，车位是公共资源，每辆车好比一个线程，看门人相当于信号量。

Pthread中的信号量

- 信号量的数据类型为结构sem_t，长整型数。
- sem_post(sem_t *sem)
  - 增加信号量的值。当有线程阻塞在这个信号量上时，调用这个函数会使其中的一个线程开始运行，选择机制由线程的调度策略决定。
- em_wait(sem_t *sem)
  - 阻塞当前线程直到信号量sem的值大于0，解除阻塞后将sem的值减一，表明公共资源经使用后减少。

互斥锁

- 互斥锁（mutex 是 MUTual EXclusion 的缩写）是实现线程间同步的一种方法。
- 互斥锁是一种锁，线程对共享资源进行访问之前必须先获得锁；否则线程将保持等待状态，直到该锁可用。只有其他线程都不占有它时一个线程才可以占有它，在该线程主动放弃它之前也没有另外的线程可以占有它。占有这个锁的过程就叫做锁定或者获得互斥锁。

互斥锁

```
thread A
void someMethod()
{
      print("A Hello one");
      print("A Hello two");
}
```

```
thread B
void someMethod()
{
      print("B Hello one");
      print("B Hello two");
}
```

- 两个线程A 和B，如果不加任何同步原语的话，线程A和B 的输出将产生交错，即可能产生类似

```
A Hello one
B Hello one
B Hello two
A Hello one
```

互斥锁

```
thread A
void someMethod()
{
      mutex.lock();
      print("A Hello one");
      print("A Hello two");
      mutex.unlock();
}
```

```
thread B
void someMethod()
{
      mutex.lock();
      print("B Hello one");
      print("B Hello two");
      mutex.unlock();
}
```

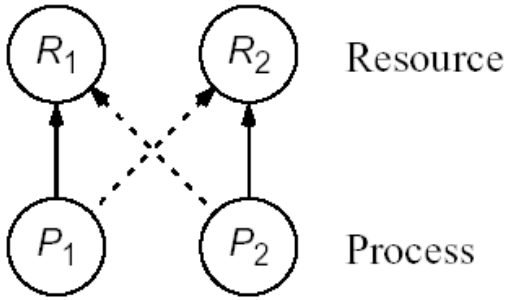
- 可能的输出结果：

```
A Hello one
A Hello two
B Hello one
B Hello two
```

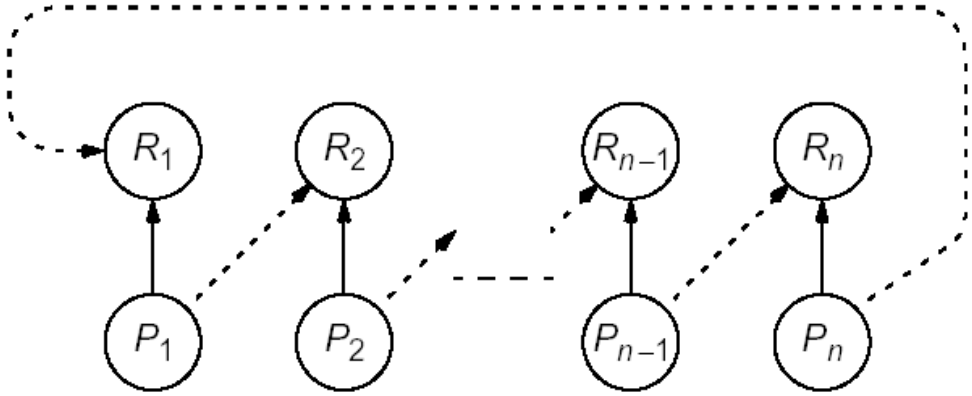
或

```
B Hello one
B Hello two
A Hello one
A Hello two
```

死锁



(a) Two-process deadlock



(b) n-process deadlock

条件变量

- 条件变量(Condition variable)是用来通知共享数据状态信息的。当特定条件满足时，线程等待或者唤醒其他合作线程。
- 条件变量不提供互斥，需要一个互斥锁来同步对共享数据的访问。

条件变量主要操作（pthread）

- pthread_cond_signal 使在条件变量上等待的线程中的一个线程重新开始。如果没有等待的线程，则什么也不做。如果有多个线程在等待该条件，只有一个能重启动，但不能指定哪一个。
- pthread_cond_broadcast 重启动等待该条件变量的所有线程。如果没有等待的线程，则什么也不做。

条件变量主要操作

- pthread_cond_wait 自动解锁互斥锁(如同执行了 pthread_unlock_mutex)，并等待条件变量触发。这时线程挂起，不占用 CPU 时间，直到条件变量被触发。在调用 pthread_cond_wait 之前，应用程序必须加锁互斥锁。pthread_cond_wait 函数返回前，自动重新对互斥锁加锁(如同执行了 pthread_lock_mutex)。
- 互斥锁的解锁和在条件变量上挂起都是自动进行的。因此，在条件变量被触发前，如果所有的线程都要对互斥锁加锁，这种机制可保证在线程加锁互斥锁和进入等待条件变量期间，条件变量不被触发。

条件变量示例

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;/*初始化互斥锁*/
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;/*初始化条件变量*/

void *thread1(void *);
void *thread2(void *);

int i=1; //全局变量
int main(void)
{
    pthread_t t_a;
    pthread_t t_b;

    pthread_create(&t_a,NULL,thread2,(void *)NULL);/*创建进程t_a*/
    pthread_create(&t_b,NULL,thread1,(void *)NULL); /*创建进程t_b*/
    pthread_join(t_b, NULL);/*等待进程t_b结束*/
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);
    exit(0);
}
```

条件变量示例

```
void *thread1(void *junk)
{
   for(i=1;i<=9;i++)
   {
      pthread_mutex_lock(&mutex);/*锁住互斥锁*/
      if(i%3==0)
         pthread_cond_signal(&cond);/*条件改变，发送信号，通知t_b进程*/
      else
         printf("thead1:%d\n",i);
      pthread_mutex_unlock(&mutex);/*解锁互斥锁*/
      sleep(1);
}}

void *thread2(void *junk)
{
   while(i<9)
   {
      pthread_mutex_lock(&mutex);
      if(i%3!=0)
         pthread_cond_wait(&cond,&mutex);/*等待*/
      printf("thread2:%d\n",i);
      pthread_mutex_unlock(&mutex);
      sleep(1);
}}
```

输出结果：

```
thread1:1
thread1:2
thread2:3
thread1:4
thread1:5
thread2:6
thread1:7
thread1:8
thread2:9
```

Outline

- 多线程基本概念
  - 线程与进程的区别
  - 线程的生命周期
  - 线程同步
- **Pthread多线程**
- 实例分析
  - 计算数组中“3”出现的次数

POSIX Thread API

- POSIX ： Portable Operating System Interface
- POSIX 是基于UNIX 的，这一标准意在期望获得源代码级的软件可移植性。为一个POSIX 兼容的操作系统编写的程序，应该可以在任何其它的POSIX 操作系统（即使是来自另一个厂商）上编译执行。
- POSIX 标准定义了操作系统应该为应用程序提供的接口：系统调用集。
- POSIX是由IEEE（Institute of Electrical and Electronic Engineering）开发的，并由ANSI（American National Standards Institute）和ISO（International Standards Organization）标准化。

程序示例

```c
#include <pthread.h>

/*
 * The function to be executed by the thread should take a
 * void* parameter and return a void* exit status code.
 */
void *thread_function(void *arg)
{
  // Cast the parameter into what is needed.
  int *incoming = (int *)arg;

  // Do whatever is necessary using *incoming as the argument.

  // The thread terminates when this function returns.
  return NULL;
}

int main(void)
{
  pthread_t thread_ID;
  void      *exit_status;
  int       value;

  // Put something meaningful into value.
  value = 42;

  // Create the thread, passing &value for the argument.
  pthread_create(&thread_ID, NULL, thread_function, &value);

  // The main program continues while the thread executes.

  // Wait for the thread to terminate.
  pthread_join(thread_ID, &exit_status);

  // Only the main thread is running now.
  return 0;
}
```

Pthread线程操作函数

POSIX 函数	描述
pthread_cancel	终止另一个线程
pthread_create	创建一个线程
pthread_detach	设置线程以释放资源
pthread_equal	测试两个线程 ID 是否相等
pthread_exit	退出线程，而不退出进程
pthread_join	等待一个线程
pthread_self	找出自己的线程 ID

pthread_creat()

pthread_create()

```
int pthread_create(                          // create a new thread
  pthread_t *tid,                            // thread ID
  const pthread_attr_t *attr,                // thread attributes
  void *(*start_routine)(void *),            // pointer to function to execute
  void *arg                                  // argument to function
);
```

Arguments:

- The thread ID of the successfully created thread.
- The thread's attributes, explained below; the `NULL` value specifies default attributes.
- The function that the new thread will execute once it is created.
- An argument passed to the `start_routine()`.

Return value:

0 if successful. Error code from `<errno.h>` otherwise.

Notes:

Use a structure to pass multiple arguments to the start routine.

pthread_join()

pthread_join()

```
int pthread_join(                    // wait for a thread to terminate
  pthread_t tid,                     // thread ID to wait for
  void **status                      // exit status
);
```

Arguments:

- The ID of the thread to wait for.
- The completion status of the exiting thread will be copied into `*status` unless `status` is `NULL`, in which case the completion status is not copied.

Return value:

0 for success. Error code from `<errno.h>` otherwise.

Notes:

Once a thread is joined, the thread no longer exists, its thread ID is no longer valid, and it cannot be joined with any other thread.

pthread_self()

pthread_self()

```
pthread_t pthread_self();                    // Get my thread ID
```

Return value:

The ID of the thread that called this function.

pthread_equal()

pthread_equal()

```
int pthread_equal(                    // Test for equality
  pthread_t t1,                       // First operand thread ID
  pthread_t t2                        // Second operand thread ID
);
```

Arguments:

Two thread IDs

Return value:

- Nonzero if the two thread IDs are the same (following the C convention).
- 0 if the two threads are different.

pthread_exit()

void pthread_exit()

```
void pthread_exit(                          // terminate a thread
  void *status                              // completion status
);
```

Arguments:

The completion status of the thread that has exited. This pointer value is available to other threads.

Return value:

None.

Notes:

When a thread exits by simply returning from the start routine, the thread's completion status is set to the start routine's return value.

pthread_cancle()

```
int pthread_cancel(pthread_t thread);
```

- 参数thread 是要取消的目标线程的线程ID。该函数并不阻塞调用线程，它发出取消请求后就返回。如果成功，pthread_cancel 返回0，如果不成功，pthread_cancel 返回一个非零的错误码。
- 线程收到一个取消请求时会发生什么情况取决于它的状态和类型。如果线程处于PTHREAD_CANCEL_ENABLE 状态，它就接受取消请求，如果线程处于PTHREAD_CANCEL_DISABLE状态，取消请求就会被保持在挂起状态。默认情况下，线程处于PTHREAD_CANCEL_ENABLE状态。

pthread_detach()

```
int pthread_detach(pthread_t thread)
```

- 设置线程的内部选项来说明线程退出后，其所占有的资源可以被回收。参数thread是要分离的线程的ID。被分离的的线程退出时不会报告它们的状态。
- 如果函数调用成功，pthread_detach 返回0，如果不成功，pthread_detach 返回一个非零的错误码。

错误	原因
EINVAL	thread 对应的不是一个可分离的线程.
ESRCH	没有 ID 为 thread 的线程

Pthread 锁例程

- 在 Pthreads 中，锁实现为 *mutually exclusive lock* 变量或者“mutex” 变量。
- 要使用 mutex，必须将其声明为 *pthread_mutex_t* 类型，并初始化：

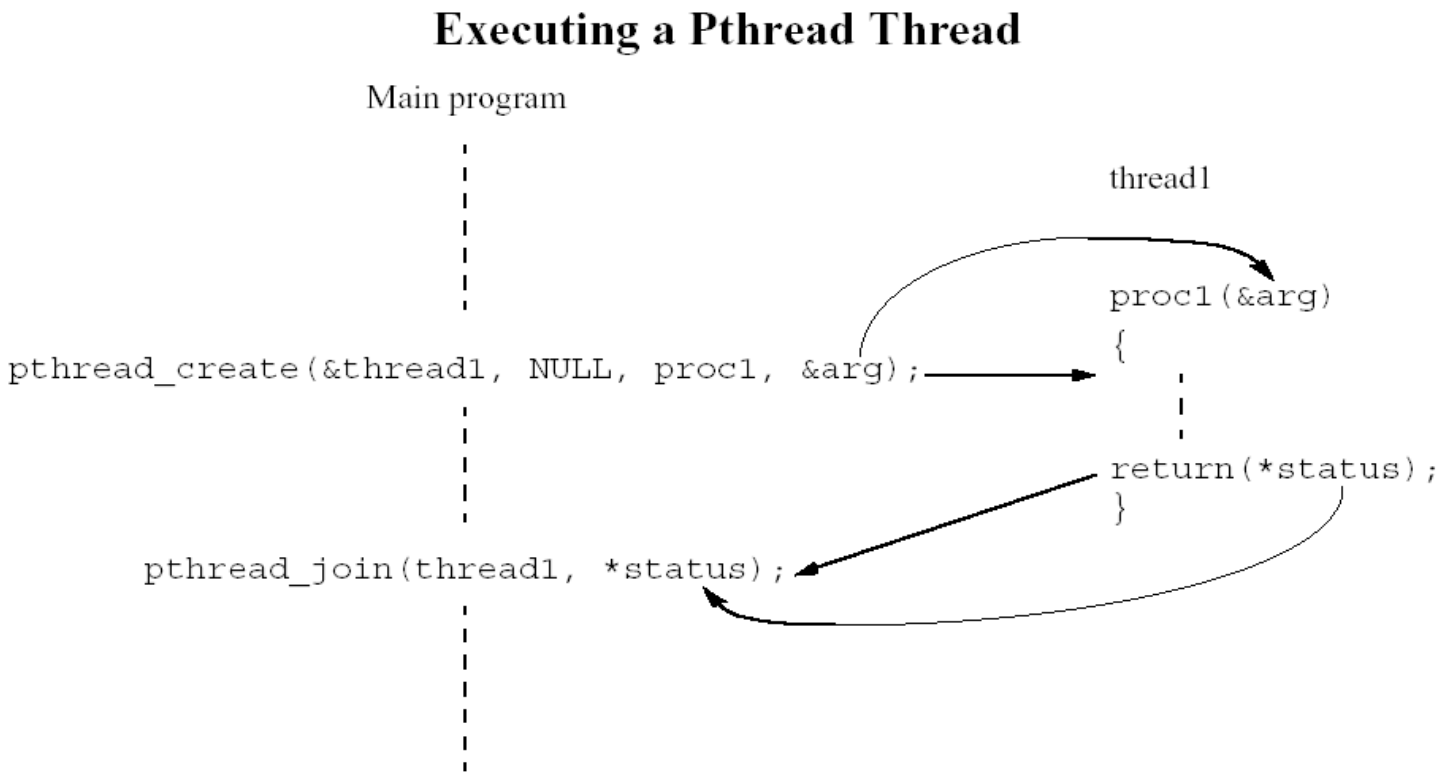
```
pthread_mutex_t mutex1;
        .
        .
pthread_mutex_init(&mutex1, NULL);
```

```
pthread_mutex_lock(&mutex1);
        .
   critical section
        .
pthread_mutex_unlock(&mutex1);
```

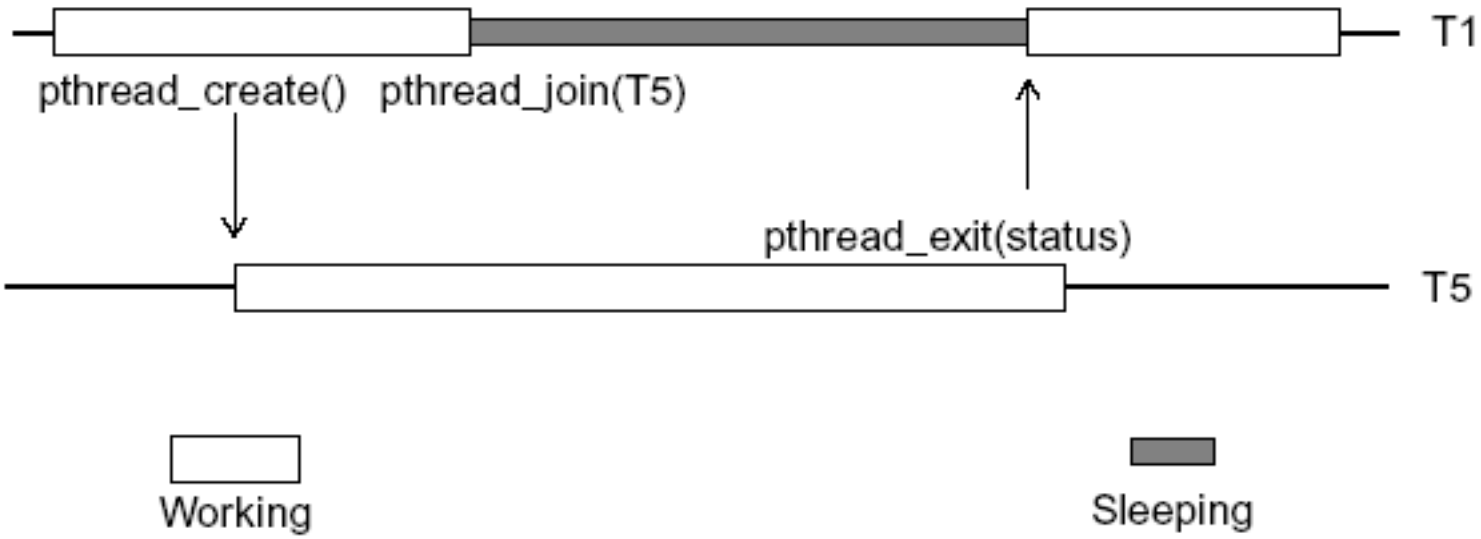
- 使用 *pthread_mutex_destroy()* 销毁 mutex。

执行 Pthread 线程

IEEE Portable Operating System Interface, POSIX, section 1003.1 standard

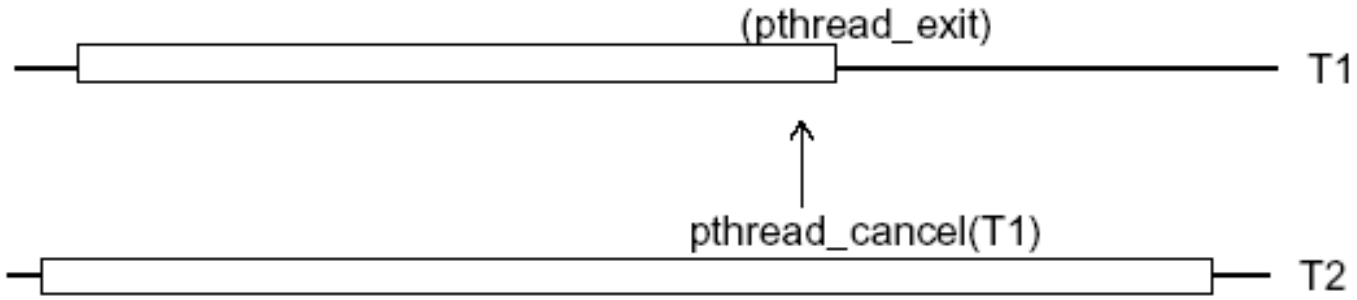


Pthread线程的生命周期



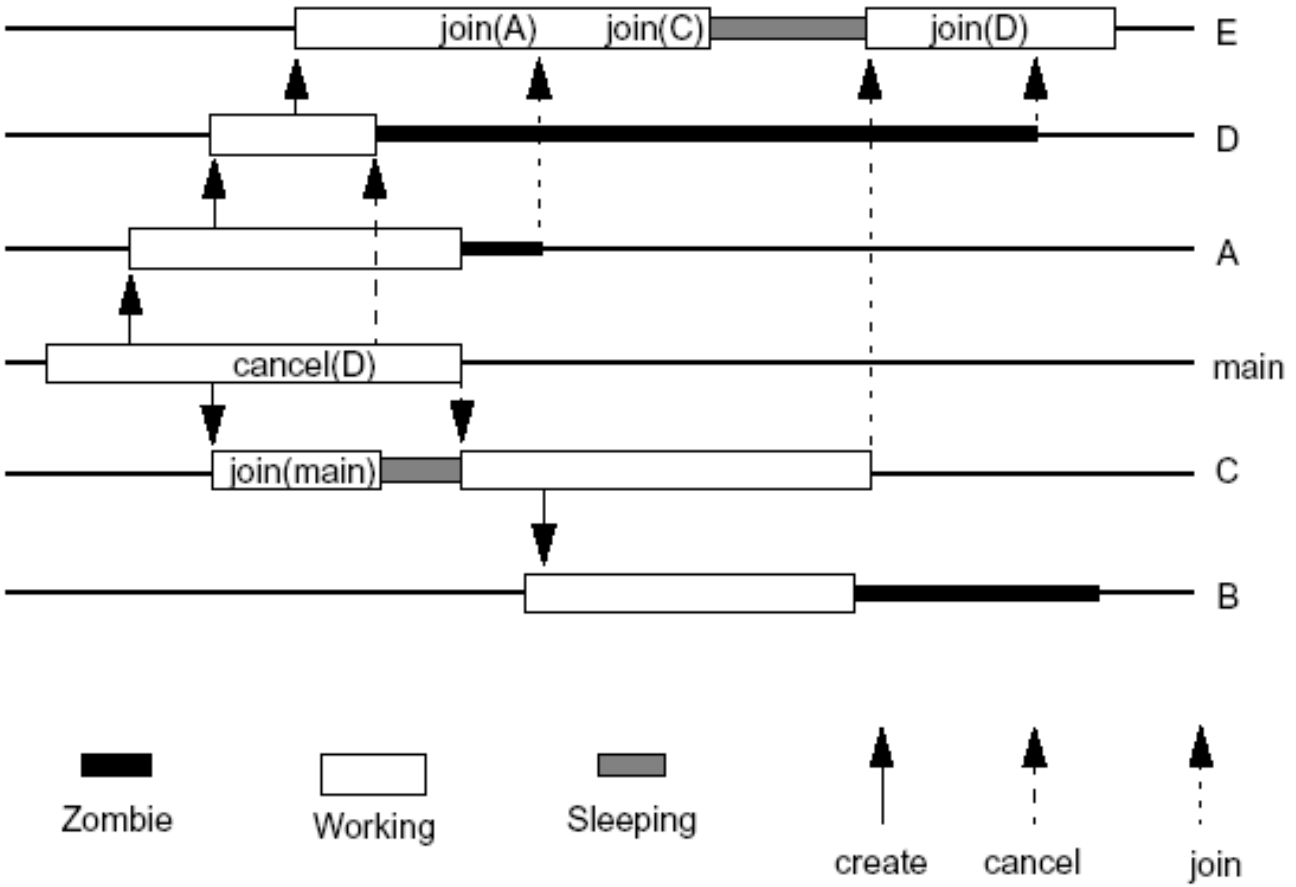
使用 pthread_join() 和 pthread_exit()

Pthread线程的生命周期（续）



一个线程被另一个线程 cancel

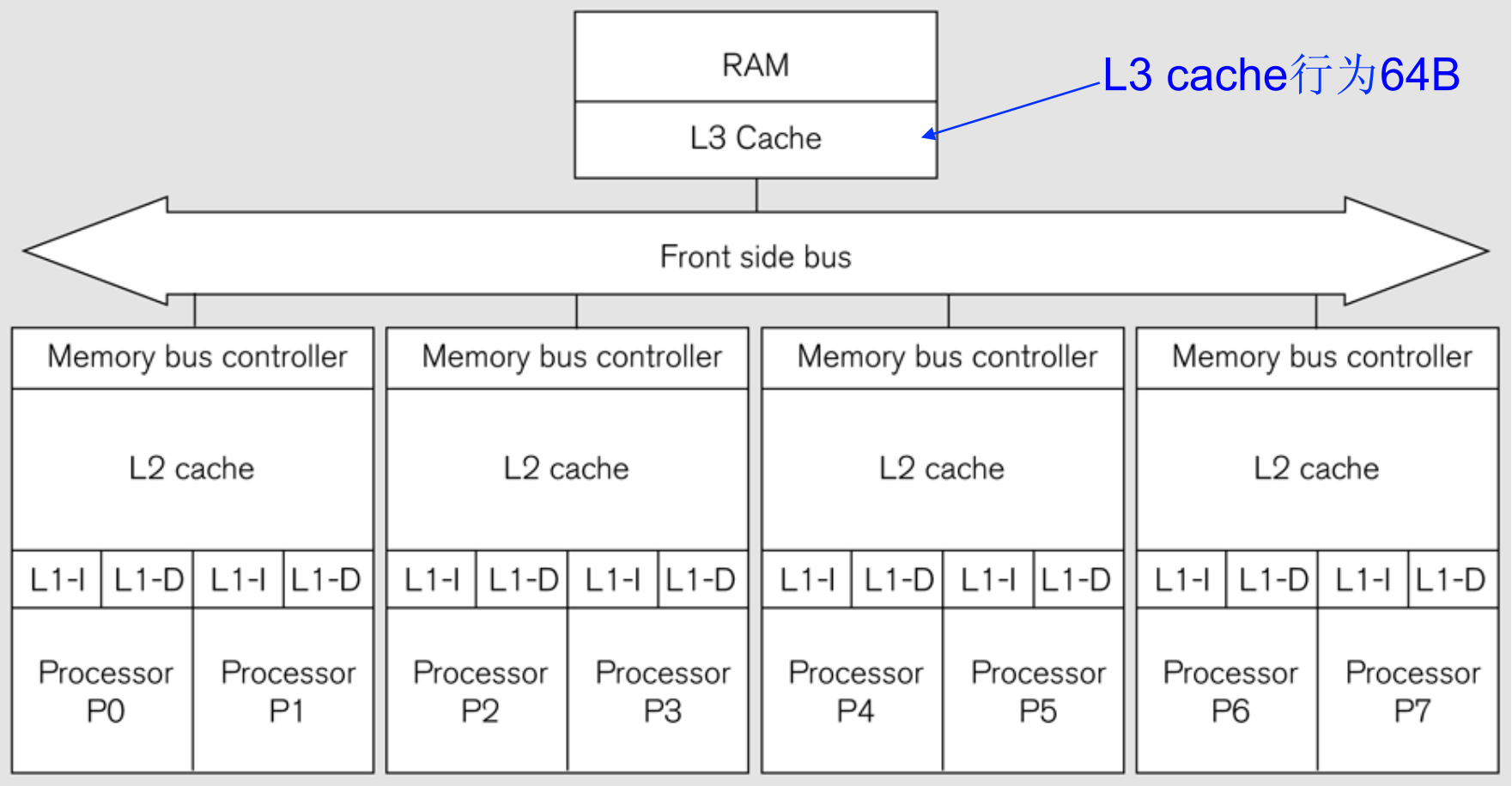
Pthread线程的生命周期（续）



Outline

- 多线程基本概念
  - 线程与进程的区别
  - 线程的生命周期
  - 线程同步
- Pthread多线程
- **实例分析**
  - **计算数组中“3”出现的次数**

实验环境中多核计算机系统结构

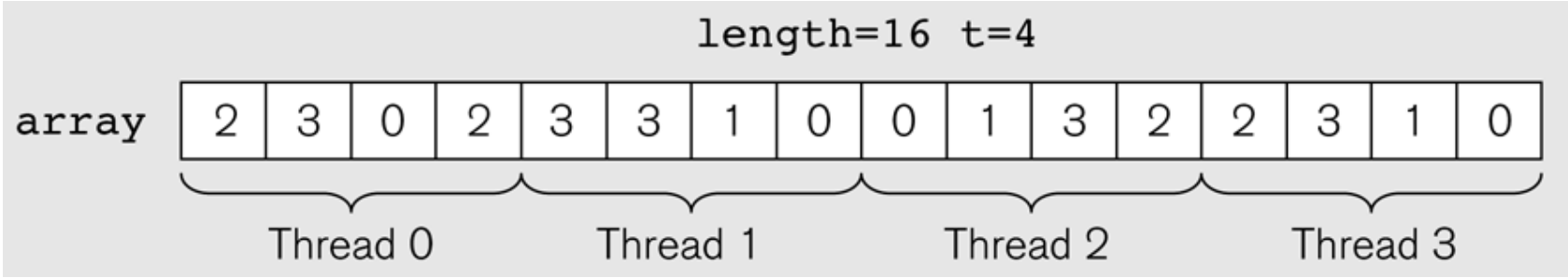


统计3的个数：串行代码（伪代码）

```
int *array;
int length;
int count;
int count3s()
{
  int i;
  count=0;
  for (i=0; i<length; i++)
  {
     if(array[i]==3
     {
      count++;
     }
  }
  return count;
}
```

对数组的划分

- 长度为16，线程数为4



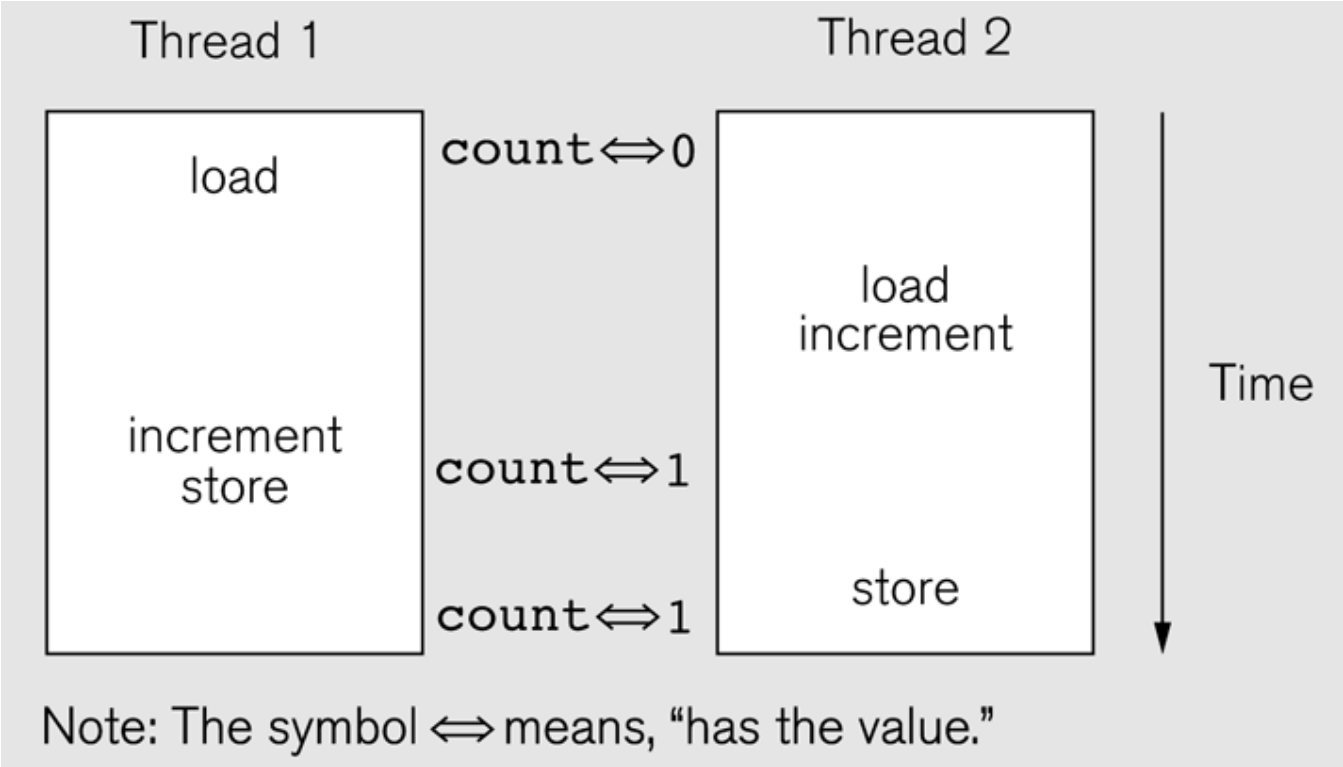
- 实际实验中数组规模为50M，随机分布30%的数值3。实验结果是1000次运行的均值。

并行算法1

此算法不能获得正确的结果

```
1  int t;                    /* number of threads */
2  int *array;
3  int length;
4  int count;
5
6  void count3s()
7  {
8    int i;
9    count = 0;
10   /* Create t threads */
11   for(i=0; i<t; i++)
12   {
13     thread_create(count3s_thread, i);
14   }
15
16   return count;
17 }
18
19 void count3s_thread(int id)
20 {
21   /* Compute portion of the array that this thread
        should work on */
22      int length_per_thread=length/t;
23      int start=id*length_per_thread;
24
25   for(i=start; i<start+length_per_thread; i++)
26   {
27     if(array[i]==3)
28     {
29       count++;
30     }
31   }
32 }
```

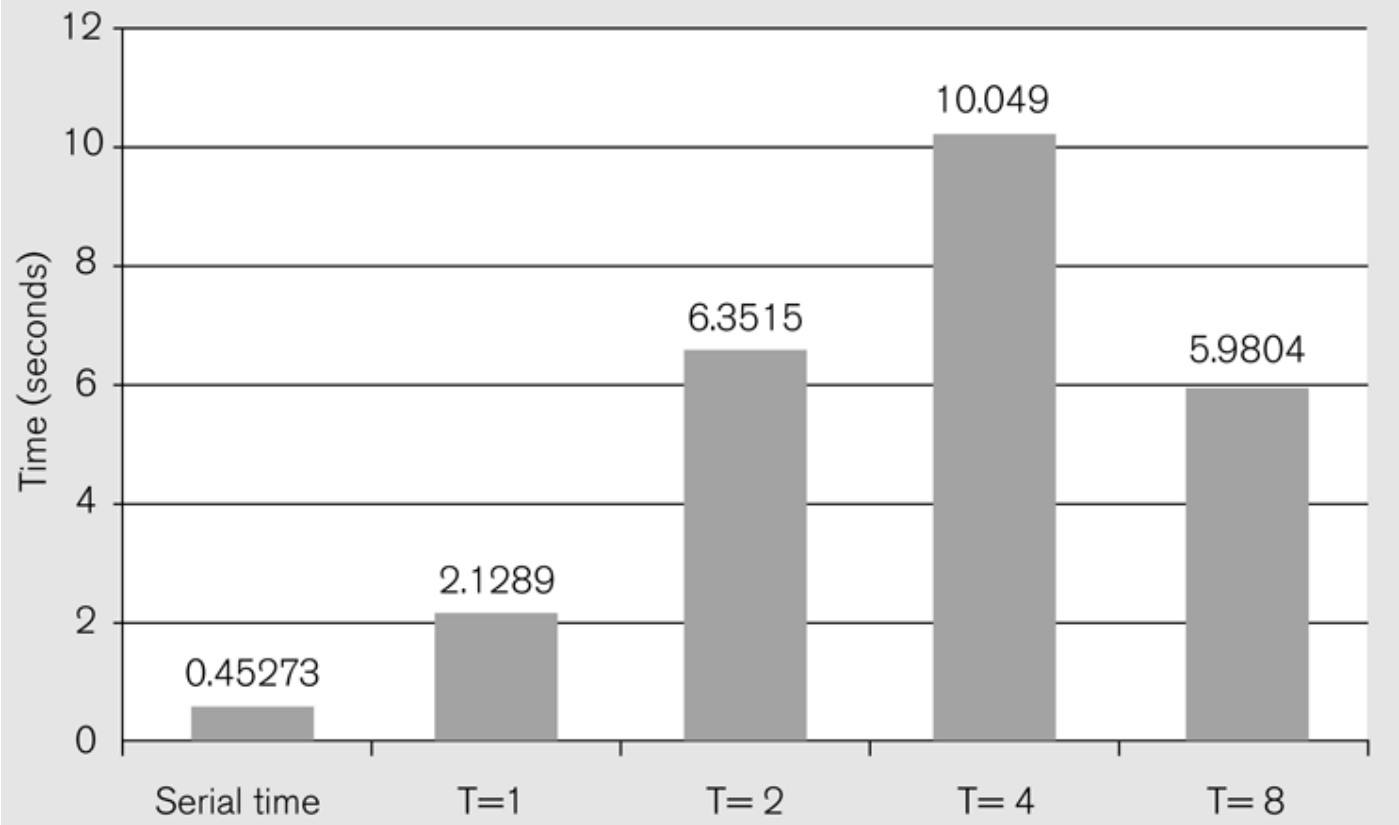
竞态条件


Thread 1
Thread 2
load
increment
store
load
increment
store
count⇔0
count⇔1
count⇔1
Time
Note: The symbol ⇔ means, "has the value."

并行算法2：加互斥锁

```
 1  mutex m;
 2
 3  void count3s_thread(int id)
 4  {
 5    /* Compute portion of the array that this thread
        should work on */
 6    int length_per_thread=length/t;
 7    int start=id*length_per_thread;
 8
 9    for(i=start; i<start+length_per_thread; i++)
10    {
11      if(array[i]==3)
12      {
13        mutex_lock(m);
14        count++;
15        mutex_unlock(m);
16      }
17    }
18  }
```

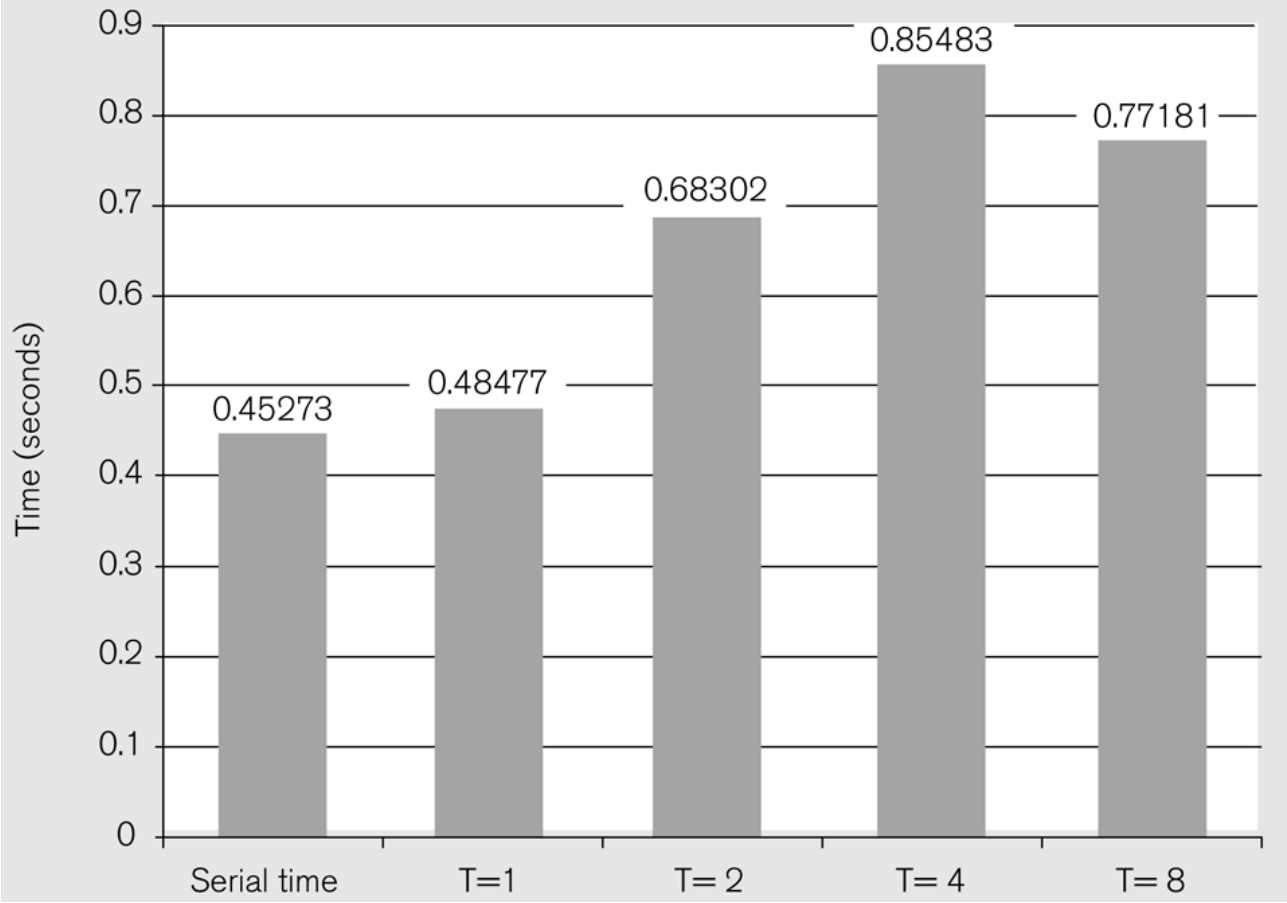
并行算法2的性能


12
10
8
6
4
2
0
Time (seconds)
0.45273
2.1289
6.3515
10.049
5.9804
Serial time
T=1
T= 2
T= 4
T= 8

并行算法3

```
 1  private_count[MaxThreads];
 2  mutex m;
 3
 4  void count3s_thread(int id)
 5  {
 6    /* Compute portion of array for this thread to
        work on */
 7    int length_per_thread=length/t;
 8    int start=id*length_per_thread;
 9
10    for(i=start; i<start+length_per_thread; i++)
11    {
12      if(array[i] == 3)
13      {
14        private_count[id]++;
15      }
16    }
17    mutex_lock(m);
18    count+=private_count[id];
19    mutex_unlock(m);
20  }
```

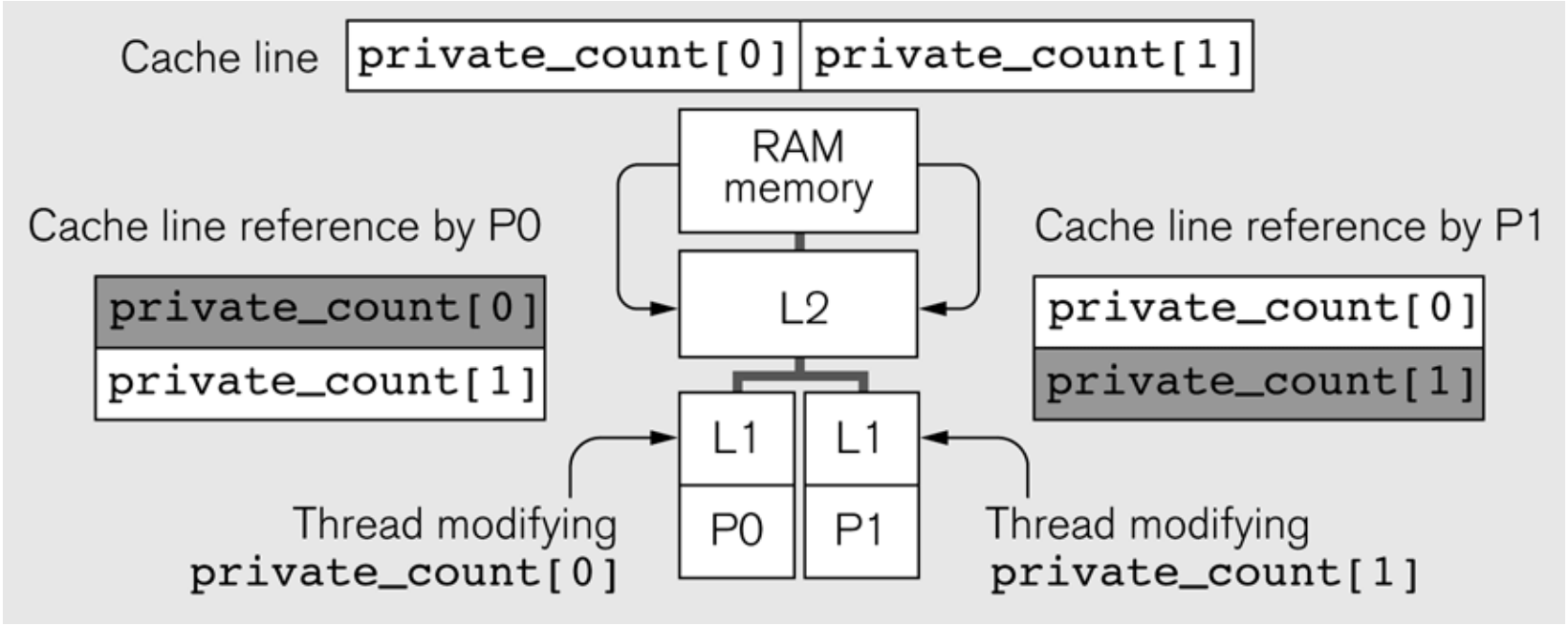
并行算法3的性能


0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Time (seconds)
0.45273
0.48477
0.68302
0.85483
0.77181
Serial time
T=1
T= 2
T= 4
T= 8

cache一致性

- cache一致性的单位是行（本例中一行为64B）
- 对cache行中的任意部分的修改等同于对整个行的修改
- L3 cache行修改后将触发L2、L1缓存的更新
- 处理器P0和P1上的线程对private_count[0]或private_count[1]进行互斥访问，但底层系统将它们置于同一个64B的cache行中。

假共享（false sharing）

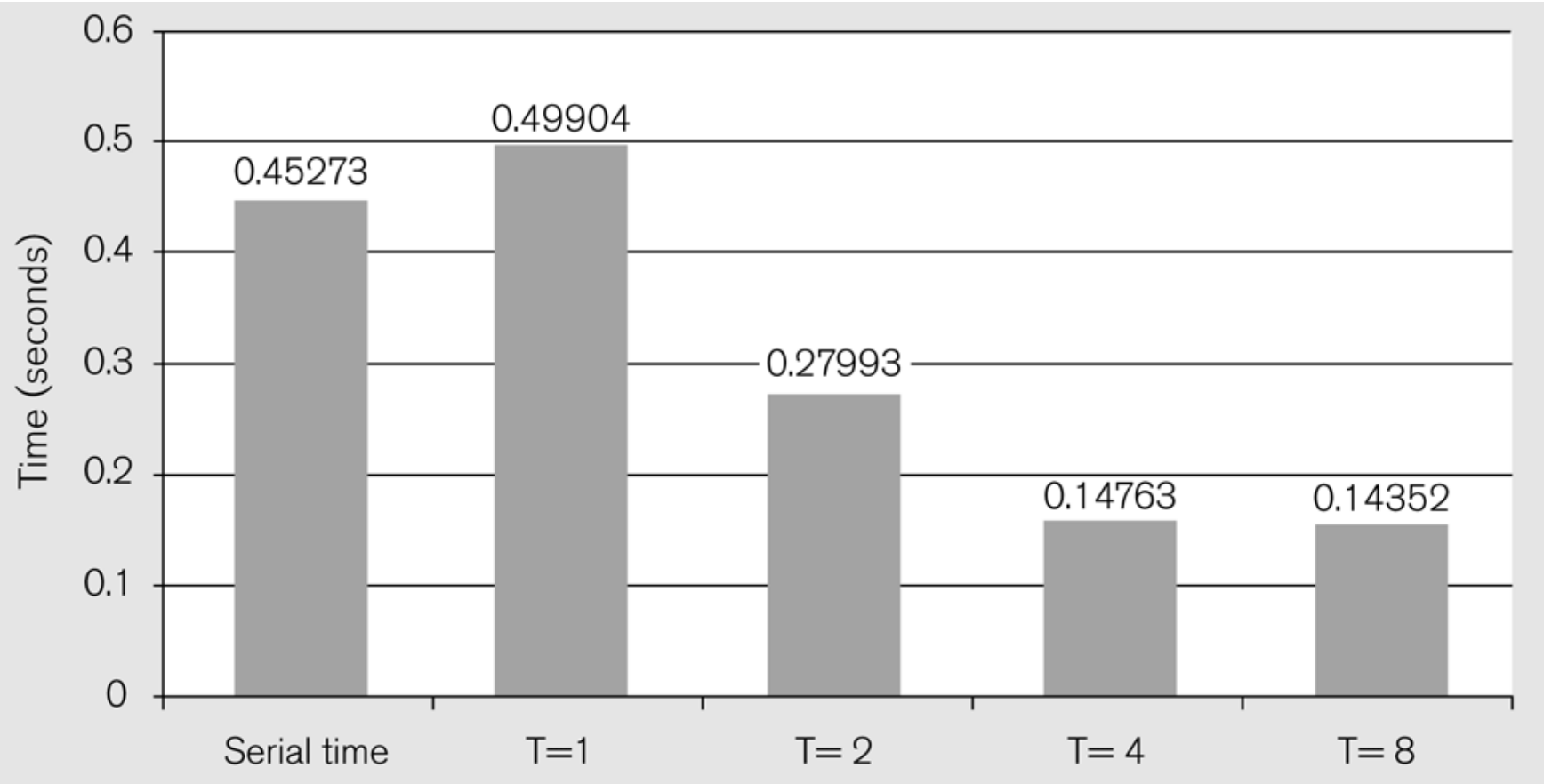


并行算法4：避免false sharing

```
 1  struct padded_int
 2  {
 3    int value;
 4    char padding[60];
 5  } private_count[MaxThreads];
 6
 7  void count3s_thread(int id)
 8  {
 9    /*  Compute portion of the array this thread should
          work on */
10    int length_per_thread=length/t;
11    int start=id*length_per_thread;
12
13    for(i=start; i<start+length_per_thread; i++)
14    {
15      if(array[i] == 3)
16      {
17        private_count[id]++; (private_count[id].value++;)
18      }
19    }
20    mutex_lock(m);
21    count+=private_count[id].value;
22    mutex_unlock(m);
23  }
```

cache行大小为64B

并行算法4的性能


0.6
0.5
0.4
0.3
0.2
0.1
0
Time (seconds)
0.45273
0.49904
0.27993
0.14763
0.14352
Serial time
T=1
T= 2
T= 4
T= 8