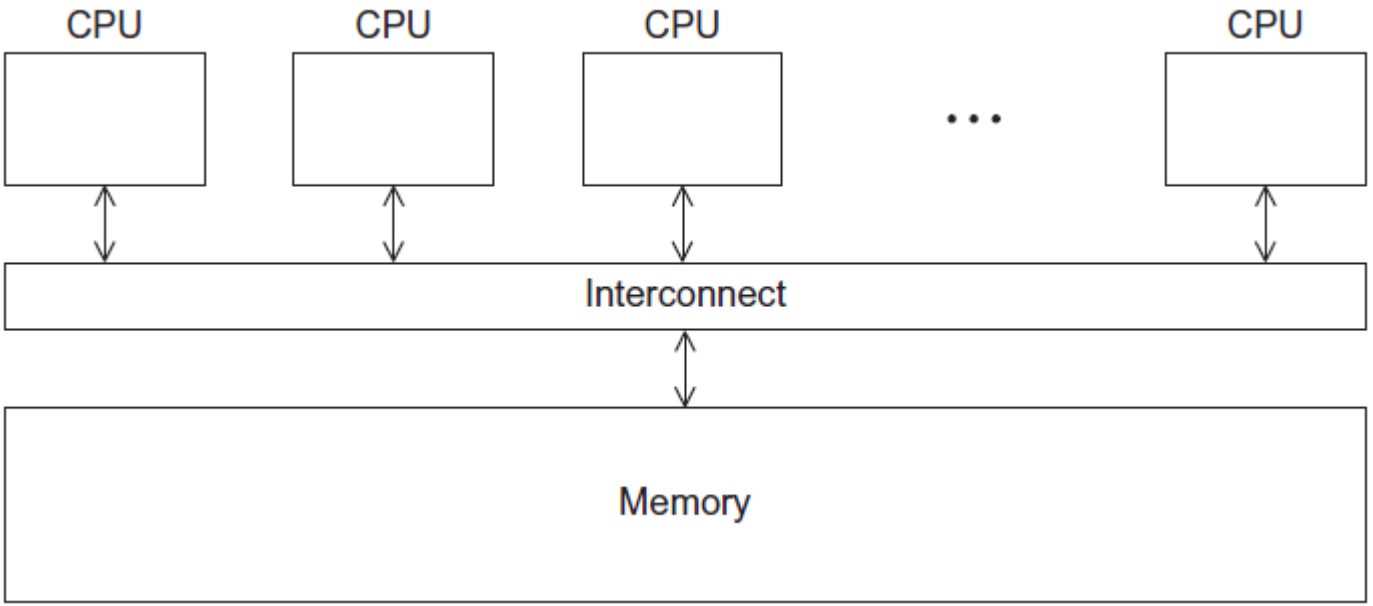
太原理工大学
TAIYUAN UNIVERSITY OF TECHNOLOGY

第4章
用Pthreads进行共享内存编程

岳俊宏
E-mail:yuejunhong@tyut.edu.cn; Tel:18234095983

目录

共享内存系统



在共享内存编程模型中，所有线程中的并行程序都可以访问内存的任何区域。

存在临界区问题。


进程、线程
和Pthreads

进程和线程

- 进程是正在运行程序的实例
- 进程可以启动多个线程，线程可以共享这个进程的大部分资源
- 线程是与进程类似的轻量级进程
- 在一个共享内存程序中，一个进程可拥有多个控制线程

POSIX线程库

- 也称为Pthreads线程库.
- POSIX是一个类Unix操作系统上的标准库，可链接到C程序中的库.
- 它定义了一套多线程编程的应用程序编程接口.

说明

- Pthreads的API只在支持POSIX的系统上才有效—Linux, MacOS X, Solaris, HPUX, …（unix系统）

Hello World 程序

Hello World程序 (1)

声明Pthreads的函数、常量和类型等

```
#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));
```

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 */
```

Hello World程序 (3)

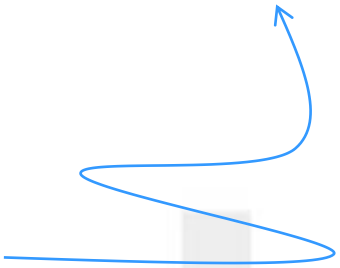
```
void *Hello(void* rank) {
   long my_rank = (long) rank;  /* Use long in case of 64-bit system */

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

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

编译Pthread程序

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



运行Pthreads程序

Running a Pthreads program

. / 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

- 可能引起令人困惑的错误
- 应该限制使用全局变量，除了确实需要用到的情况，如：

  — 线程间共享变量.

strtol函数

- strtol函数的功能是将字符串转化为long int（长整型），它在stdlib.h中声明，它的语法形式为：

```
long strtol(
const char* number_p /* in */
char** end_p /* out */
int base/* in */)
```

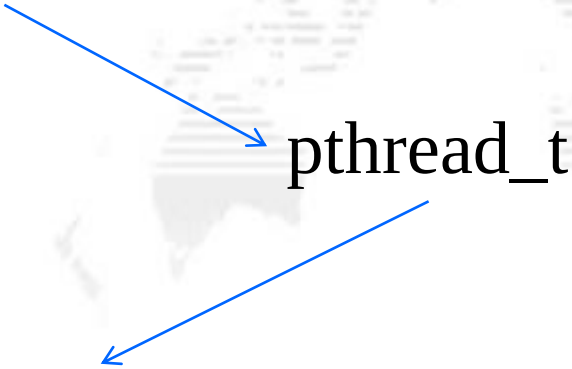
- ✓ number_p:指向的字符串转换得到的长整型数
- ✓ end_p:指向number_p字符串中第一个无效字符NULL
- ✓ base:基，10

启动线程

- MPI中进程通常是由脚本来启动的.
- Pthreads是直接由可执行程序启动的.

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
- 对象中存储的数据是系统绑定的
- 用户级代码无法直接访问到里面的数据.
- Pthreads标准保证pthread_t对象中必须存有足够多的信息，足以让pthread_t对象对它所从属的线程进行唯一标识.
- Pthread_create可以让线程取得它的专有pthread_t对象

生成线程

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

不使用该参数，所以只是在函数调用时把NULL传递给参数

在调用该函数之前就为pthread_t对象分配内存空间

生成线程

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

指向传递给函数start_routine的参数

线程将要运行的函数

运行函数(pthread_create)

- 原型:

```
void*  thread_function ( void*  args_p ) ;
```

- void* 可以转换为任意指针类型
- args_p能够指向一个列表，包含一个或多个参数值.
- 函数返回值也指向一个列表.

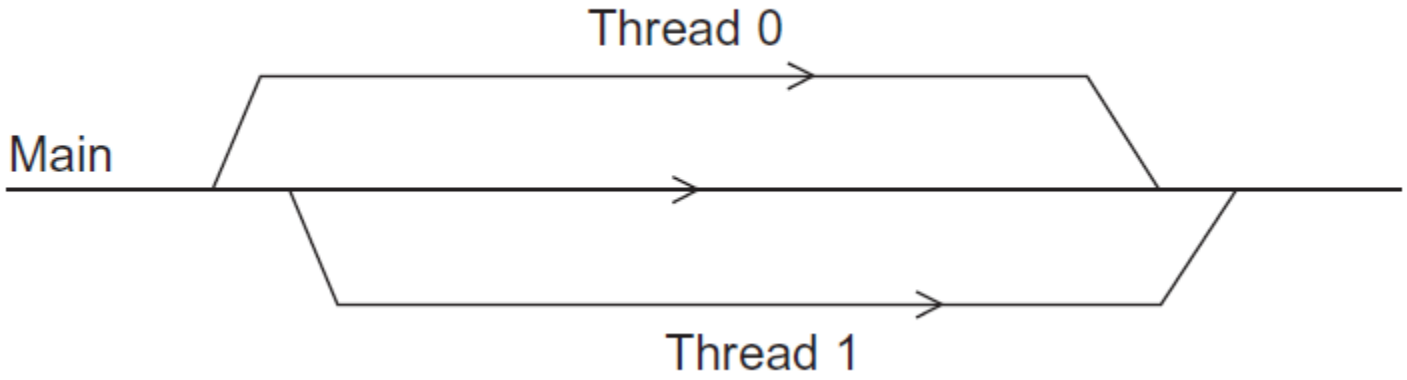
运行函数(pthread_create)

- **将函数名(地址)作为参数传递：**

```
#include<stdio.h>
int add(int a,int b){return a+b;}
int AAA(int a,int b, int(*p)(int,int) ) //注意定义的函数指针
{return p(a,b);//通过函数指针p调用所指向函数}
int main(){
int a=1,b=2;
printf("%d",AAA(a,b,add));//将函数名（函数地址）作为参数
return 0;
}
```

注意：在函数AAA中对函数指针的声明，声明之后，p就可以作为一个函数指针，该函数指针能指向参数符合（int，int）形式的所有函数，并且可以直接通过p函数指针访问所指向函数。

运行线程



主线程派生与合并两个线程

停止线程

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

矩阵-向量乘法

问题描述

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{bmatrix}$$

利用PTHREADS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];
}
```

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

使用3个线程

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];
```

每个线程需要访问x的每个元素，x需设为共享向量

矩阵-向量乘法

- 将A与y都设为共享：
  - ✓ 主函数就可以简单地通过读取标准输入stdin来初始化矩阵A；
  - ✓ 向量y也很容易被主线程打印输出。
- 只有共享变量y被改写：
  - ✓ 每个线程各自改变属于自己运算的那一部分，没有两个或两个以上线程共同处理同一部分的情况

矩阵-向量乘法并行代码

```
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-向量乘法并行代码

```
/* Global variables */
int     thread_count;
int     m, n;
double* A;
double* x;
double* y;
```

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

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

Print_vector("The product is", y, m);
```

举例

```
[hdusr@Node1 ch4]$ ./pth_mat_vect 3
Enter m and n
3 3
Enter the matrix
1 2 3
4 5 6
7 8 9
We read
 1.0  2.0  3.0
 4.0  5.0  6.0
 7.0  8.0  9.0
Enter the vector
1 2 3
We read
 1.0  2.0  3.0
The product is
14.0 32.0 50.0
```

思考

如果多个线程需要更新同一内存单元的数据会怎样?


临界区

计算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 */
```

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

Note that as we increase n, the estimate with one thread gets better and better.

可能的竞争条件

- 假设有两个进程，每个线程计算自己的私有变量y，并将其加到共享变量x中，主线程将x设为0.

```
y=Comput(my_rank);
x=x+y
```

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

多个进程尝试更新同一个共享变量

临界区

- 当多个线程都要访问共享变量或共享文件时，如果至少其中一个访问是更新操作，那么这种访问就可能导致某种错误，称之为竞争条件。
- 临界区是一个更新共享资源的代码段，一次只允许一个线程执行该代码段。

例如：x=x+y

解决方案

- 忙等待
- 互斥量
- 信号量

忙等待

```
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 */
```

忙等待

```
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;
```

忙等待

```
    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 */
```

最小化临界区的执行次数

因为处于忙等待的线程仍然在持续使用CPU，所忙等待不是限制临界区访问的最理想方法。

互斥量

互斥量是互斥锁的简称，它是一个特殊类型的变量，通过某些特殊类型的函数，互斥量可以用来限制每次只有一个线程能进入临界区。

```
pthread_mutex_init(&head_mutex, NULL);//初始化
pthread_mutex_destroy(&head_mutex);//销毁
pthread_mutex_lock(&head_mutex);//获得临界区的访问权
pthread_mutex_unlock(&head_mutex);//退出临界区
```

互斥量

```
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 */
```

上机实验

- 编写运行书上的Pthreads “Hello, World”程序

结束!