



高性能计算技术

第八讲 共享存储编程

kjhe@scut.edu.cn

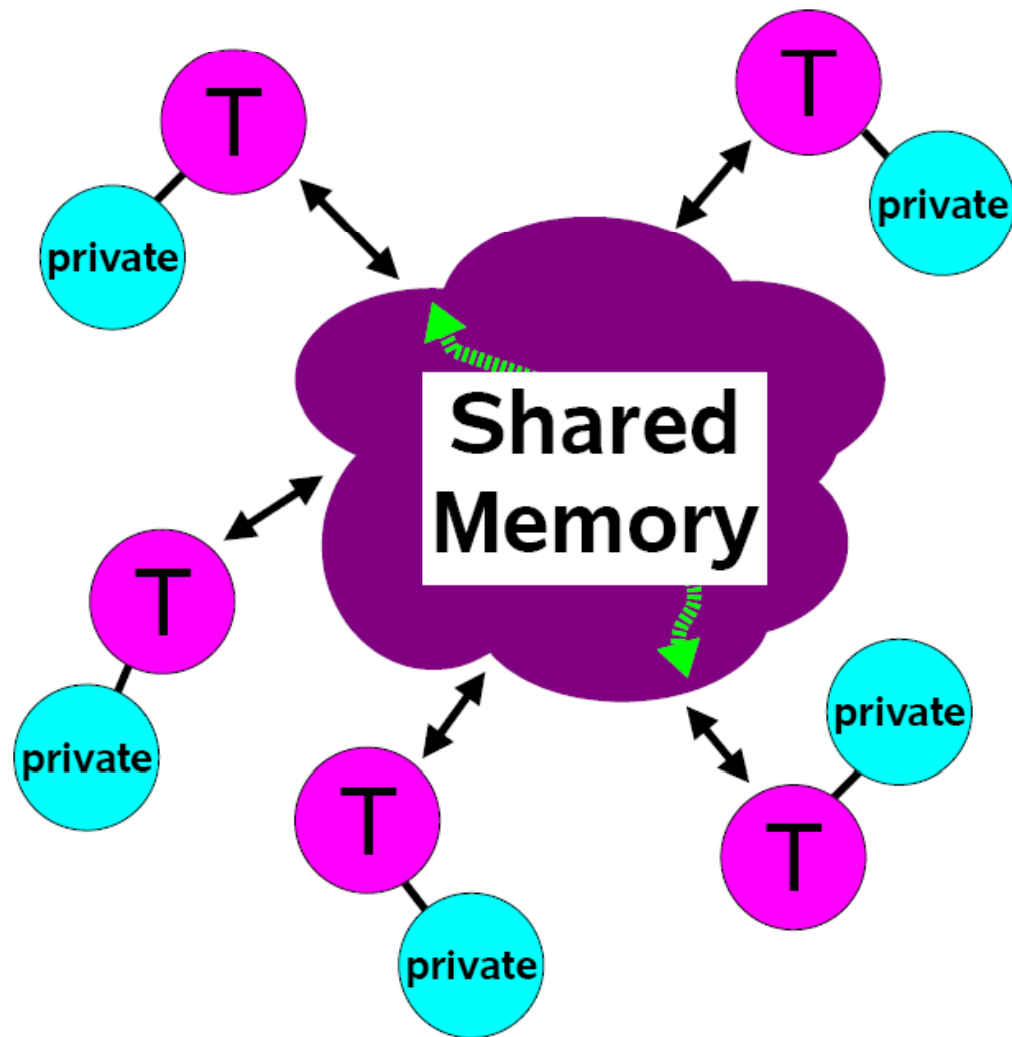
华南理工大学计算机学院

复习

- **Cannon**算法与简单分块并行算法相比，优点是什么？
- **DNS**算法的时间步长是多少？为什么可以达到？
- 什么是**Fork-Join**模型？
- 什么是共享存储编程模型？共享存储编程模型的主要特征？

共享存储编程模型

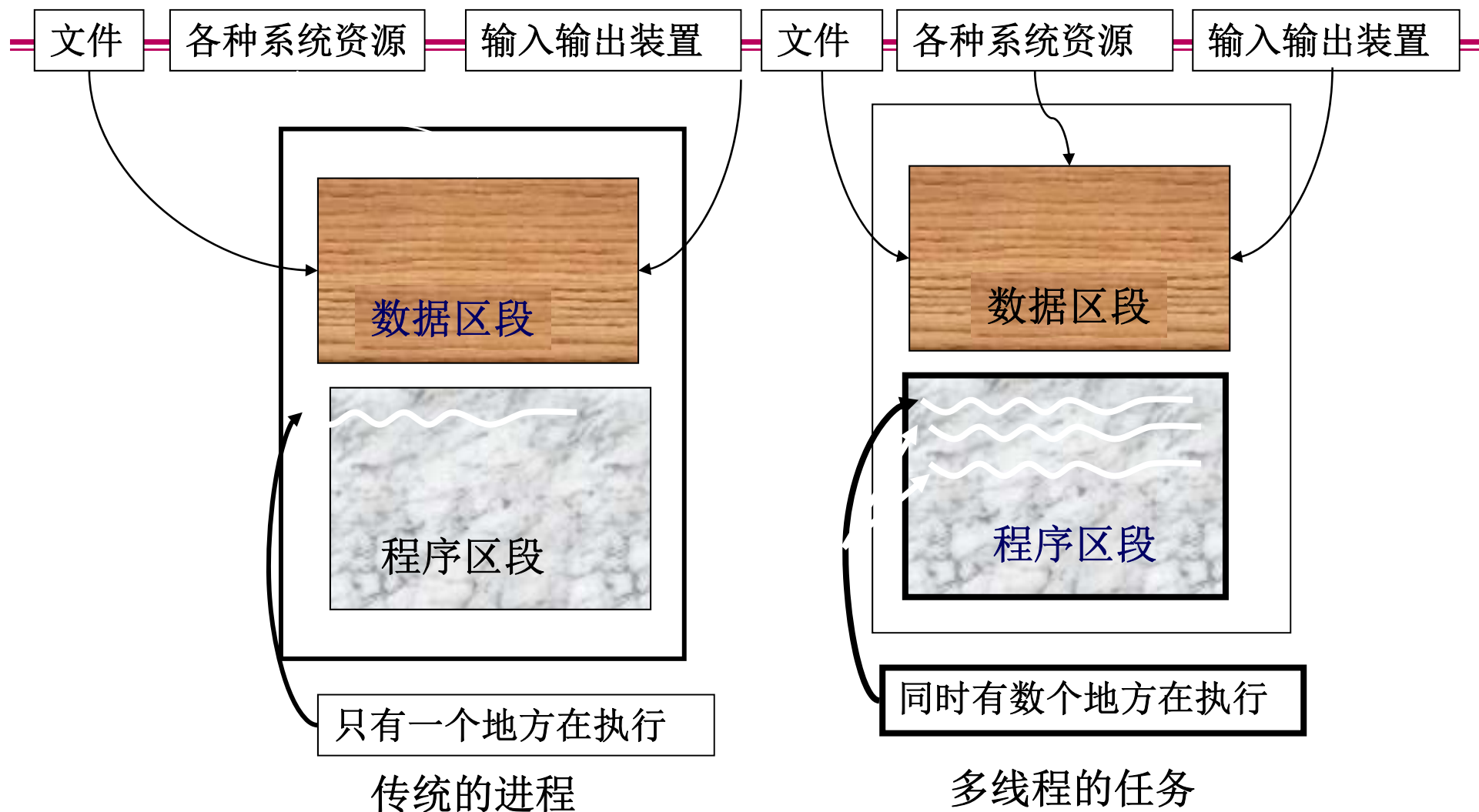
- 所有线程可以存取相同的、全局共享的存储空间
- 数据可以是共享的(shared)或私有的(private)
- 共享数据可以被所有线程所存取
- 私有数据只能被拥有者线程所存取
- 数据传输对程序员是透明的
- 需要显式说明同步，也有许多同步是隐含的



内容概要

- 线程
- **OpenMP**

线程基本概念



线程：能单独执行的计算实体，轻量级进程

多线程的优势

- 减轻编写交互频繁、涉及面多的程序的困难
- 程序的吞吐量会得到改善
- 有多个处理器的系统，可以并发运行不同的线程（否则，任何时刻只有一个线程在运行）

线程与进程的区别

- 多个进程的內部数据和状态都是完全独立的，而多线程是共享一块内存空间和一组系统资源，有可能互相影响
- 线程本身的数据通常只有寄存器数据，以及一个程序执行时使用的堆栈，所以线程的切换比进程切换的负担要小

Java线程

- 虽然Linux、Windows、Unix等操作系统支持多线程，但若要用C或C++编写多线程程序是十分困难的，因为它们对数据同步的支持不充分
- 对线程的综合支持是Java技术的一个重要特色。它提供了thread类、监视器（monitor）和条件变量（condition variable）的技术

Java 的垃圾回收器是一个低优先级的线程。

创建线程的方法

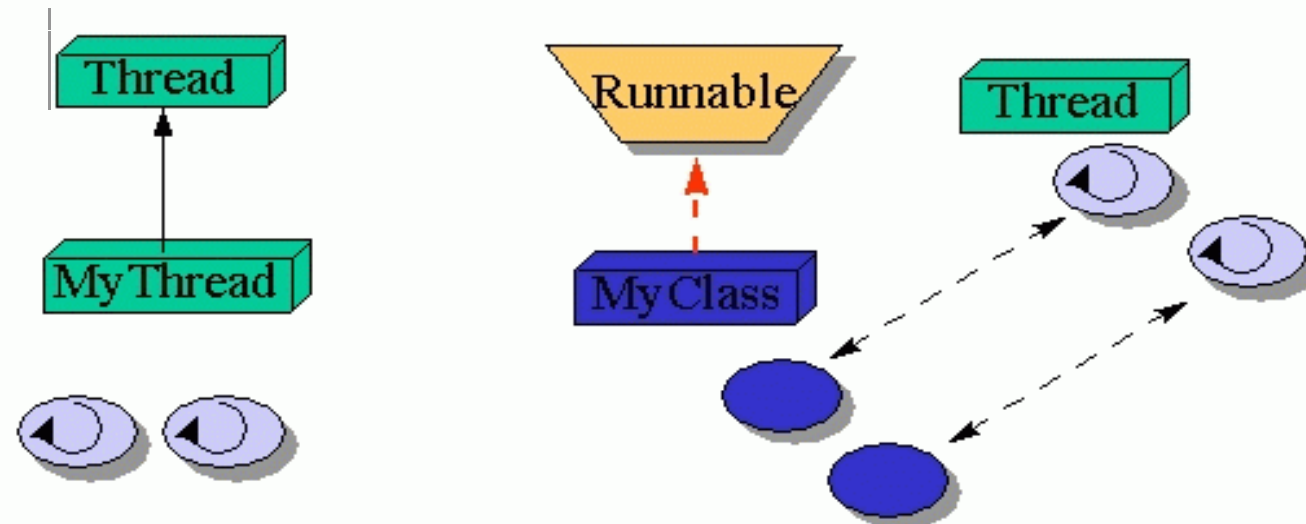
- Create a class that extends the Thread class (继承类Thread)

class MyThread extends Thread

- Create a class that implements the Runnable interface (接口)

class MyThread implements Runnable

Threading Mechanisms



创建线程

- 创建并启动线程

newthread=new Thread();

newthread.start();

- **run**方法是运行线程的主体，启动线程时，由**java**直接调用

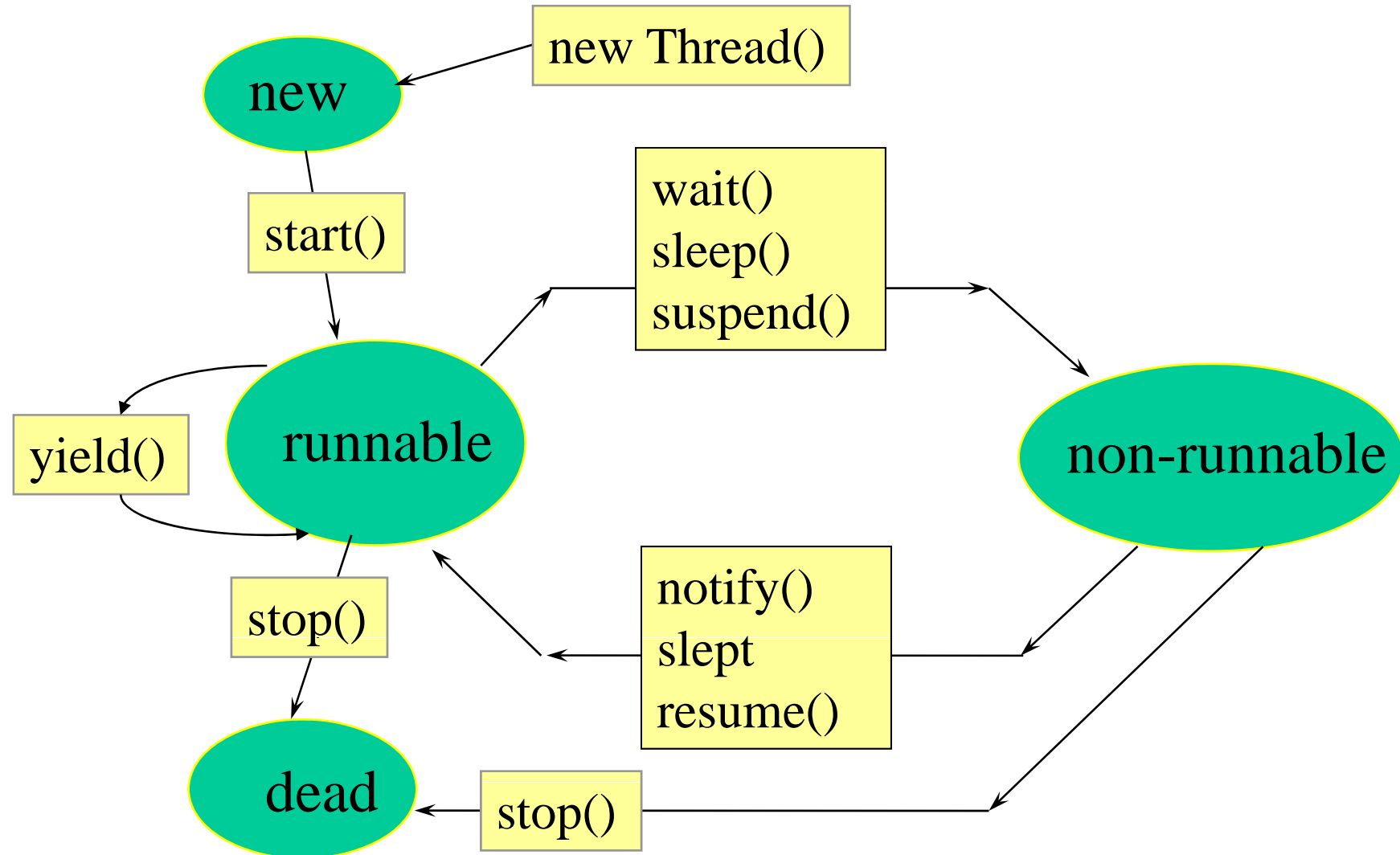
public void run()

- 停止线程，调用线程的**stop**
newthread.stop()

线程的方法

- **sleep:** 暂停线程的执行,让其它线程得到机会
- **isAlive :**判断线程目前是否正在执行状态中
 - `if(newthread.isAlive()) newthread.stop()`
- **resume:**要求被暂停得线程继续执行
- **suspend:**暂停线程的执行
- **join:** 等待线程执行完毕
 - `thatThread.join()`: 被等待的那个线程不结束,当前线程就一直等待
- **notify**和**wait**方法用来协调读取的关系
 - **notify**的作用是唤醒正在等待的线程
 - **wait**的作用是让当前线程等待
- **yield:** 将执行的权力交给其它线程, 自己到队列的最后等待

线程的生命周期



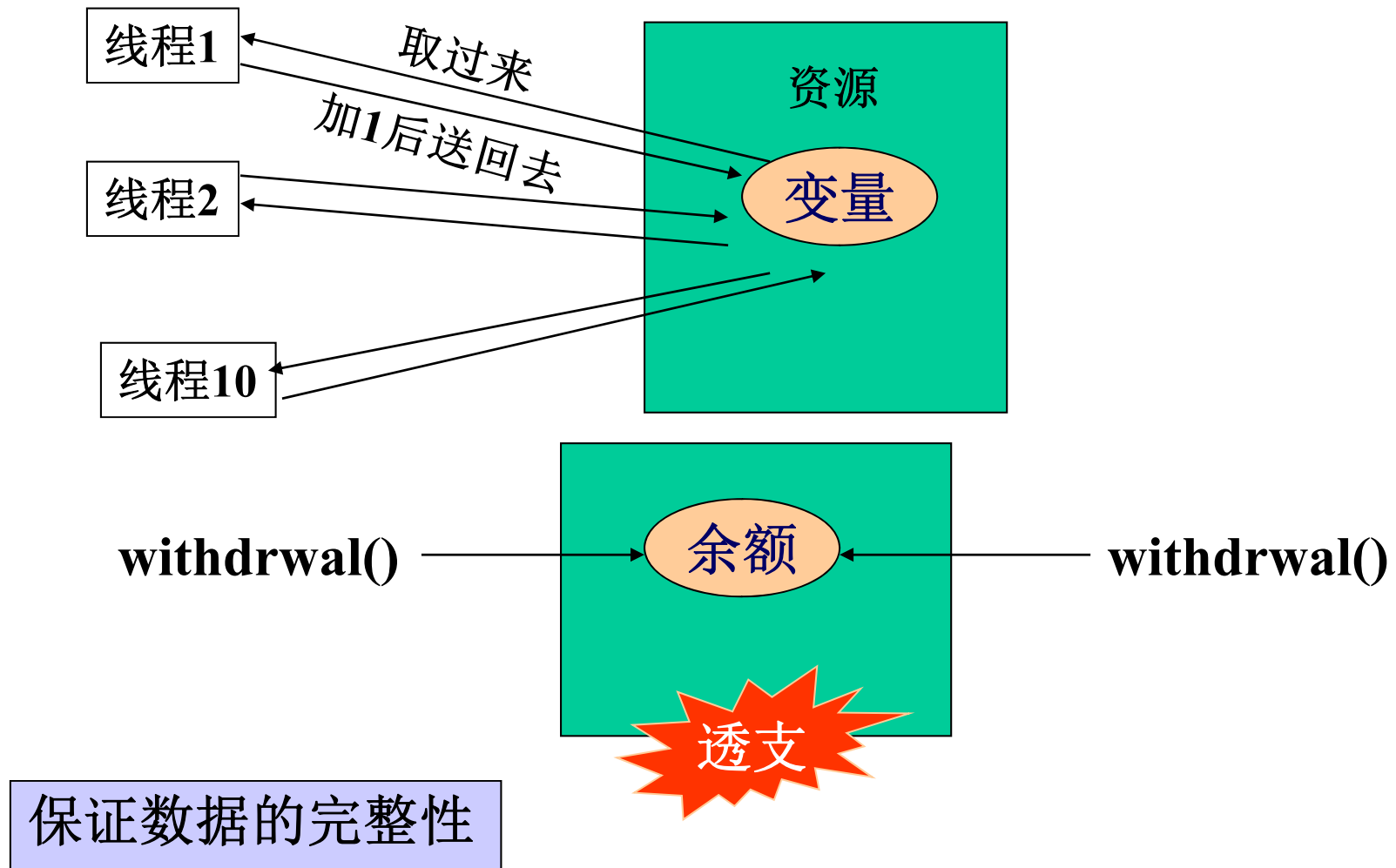
3个线程的例子

```
class myth extends Thread
{
    myth (String name)
    {
        super(name);
    }
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println("\t From "+getName()+" : i= "+i);
        }
        System.out.println("Exit from "+getName());
    }
}

class ThreadTest
{
    public static void main(String args[])
    {
        new myth("ThreadA").start();
        new myth("ThreadB").start();
        new myth("ThreadC").start();
    }
}
```

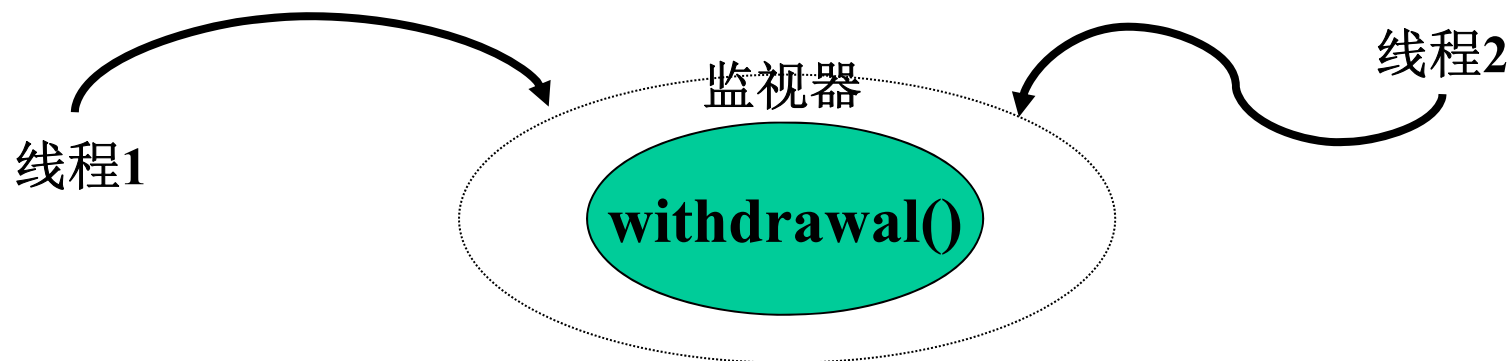
```
scutgrid11{sbdong}26:
java ThreadTest
    From ThreadA: i= 1
    From ThreadB: j= 1
    From ThreadA: i= 2
    From ThreadC: k= 1
    From ThreadB: j= 2
    From ThreadA: i= 3
    From ThreadC: k= 2
    From ThreadB: j= 3
    From ThreadA: i= 4
    From ThreadC: k= 3
    From ThreadB: j= 4
    From ThreadA: i= 5
    From ThreadC: k= 4
    From ThreadB: j= 5
Exit from A
    From ThreadC: k= 5
Exit from B
Exit from C
```

共享对象的访问



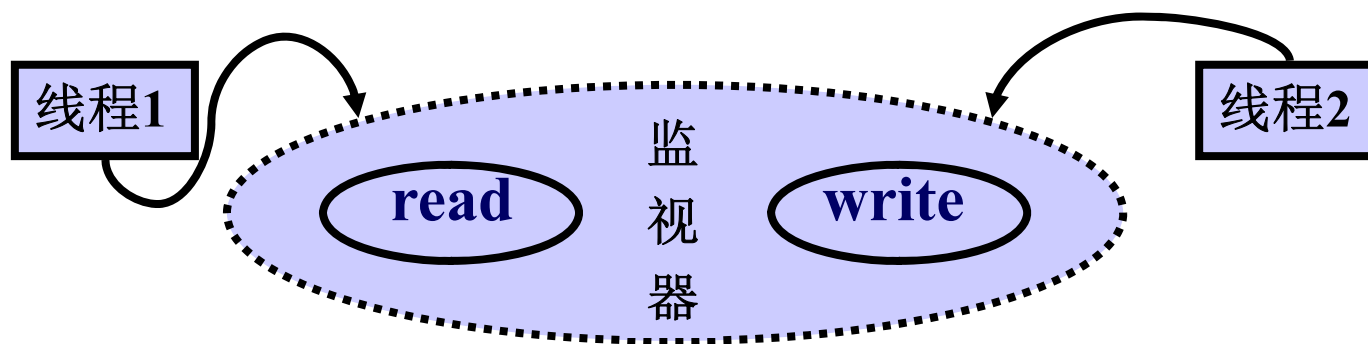
资源协调

- 对共享对象的访问必须同步，叫做**条件变量**（**condition variable**）
- **Java**语言允许通过监视器（**monitor**）使用条件变量实现线程同步
- 监视器阻止两个线程同时访问同一个条件变量。它的如同锁一样作用在数据上
- 线程1进入**withdrawal**方法时，获得监视器（加锁）；当线程1的方法执行完毕返回时，释放监视器（开锁），线程2的**withdrawal**方能进入



资源协调

- 用**synchronized**来标识的区域或方法即为监视器监视的部分。
- 一个类或一个对象有一个监视器,如果一个程序内有两个方法使用**synchronized**标志,则他们在一个监视器管理之下。

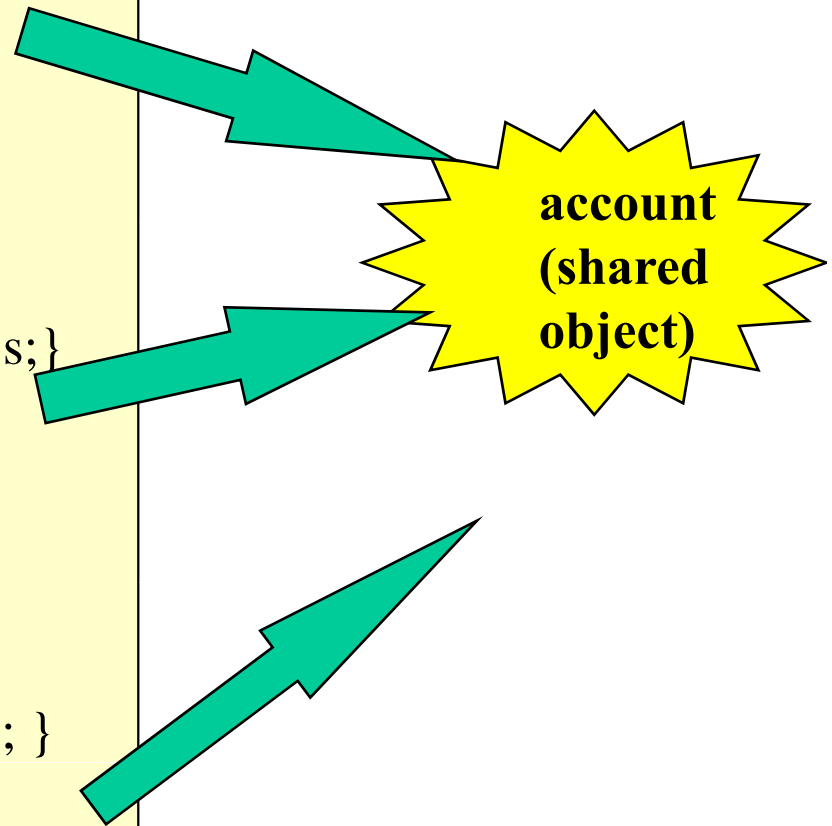


线程间的共享对象

```
class MyThread implements Runnable {  
    Account account;  
    public MyThread (Account s) { account = s;}  
    public void run() { account.deposit(); }  
} // end class MyThread
```

```
class YourThread implements Runnable {  
    Account account;  
    public YourThread (Account s) { account = s;}  
    public void run() { account.withdraw(); }  
} // end class YourThread
```

```
class HerThread implements Runnable {  
    Account account;  
    public HerThread (Account s) { account = s; }  
    public void run() { account.enquire(); }  
} // end class HerThread
```



**account
(shared
object)**

共享对象的存取例子

```
class Account { // the 'monitor'
    int balance;

    // if 'synchronized' is removed, the outcome is unpredictable
    public synchronized void deposit( ) {
        // METHOD BODY : balance += deposit_amount;
    }

    public synchronized void withdraw( ) {
        // METHOD BODY: balance -= deposit_amount;
    }
    public synchronized void enquire( ) {
        // METHOD BODY: display balance.
    }
}
```

串行化对共享对象的操作

线程的优先权

(Thread Priority)

- 多个线程运行时，调度策略为固定优先级调度。级别相同时，由操作系统按时间片来分配，即 **FCFS (First Come First Service)** 策略。
- 允许改变优先级：
 - **ThreadName.setPriority (int Number)**
 - **MIN_PRIORITY = 1**
 - **NORM_PRIORITY=5**
 - **MAX_PRIORITY=10**

线程优先级的例子

```
class ThreadPriority
{
    public static void main(String args[])
    {
        A threadA=new A();
        B threadB=new B();
        C threadC=new C();
        threadC.setPriority(Thread.MAX_PRIORITY);
        threadB.setPriority(threadA.getPriority()+1);
        threadA.setPriority(Thread.MIN_PRIORITY);
        System.out.println("Started Thread A");
        threadA.start();
        System.out.println("Started Thread B");
        threadB.start();
        System.out.println("Started Thread C");
        threadC.start();
        System.out.println("End of main thread");
    }
}
```

POSIX线程模型

- **IEEE/ANSI标准：IEEE POSIX 1003.1c-1995线程标准**
- **线程调用—线程的创建和管理（表13.1）**
 - **pthread_create**: 在进程中创建一个新线程
 - **pthread_join**: 等待其他线程结束
 - **pthread_equal**: 比较线程 id, 看是否是同一线程
 - **pthread_self**: 返回调用线程的id
 - **pthread_exit**: 结束当前运行的线程
- **线程调用—线程的同步与互斥（表13.2）**

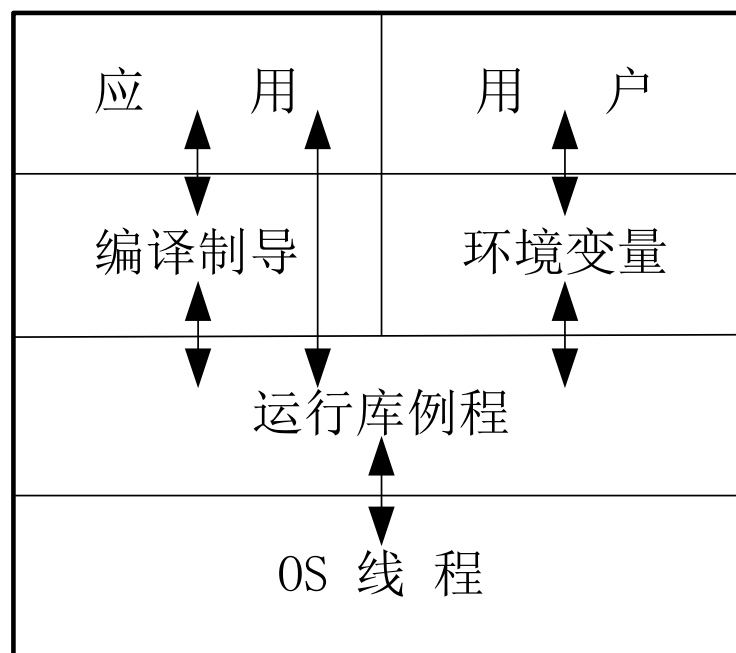
内容概要

- **OpenMP概述**
- **OpenMP编程模型**
- **OpenMP编程简介**
- **运行库例程与环境变量**
- **OpenMP计算实例**

什么是OpenMP?

- **OpenMP应用编程接口API (Application Programming Interface)** 是在共享存储体系结构上的一个编程模型
 - 三个基本API部分
 - 是C/C++ 和Fortan等的应用编程接口
 - 已经被大多数计算机硬件和软件厂家所**标准化**

OpenMP体系结构



- 三个API分量
 - 编译制导(Compiler Directives)
 - 运行库例程(Runtime Library Routines)
 - 环境变量(Environment Variables)

OpenMP的历史

- 1994年，第一个ANSI X3H5草案提出，被否决
- 1997年，OpenMP标准规范代替原先被否决的ANSI X3H5，被人们认可
- 1997年10月公布了与Fortran语言捆绑的第一个标准规范
- 1998年11月9日公布了支持C和C++的标准规范
- 当前版本3.0，2008年5月

为什么需要OpenMP?

- 早先的标准如ANSI X3H5过时，循环级并行性粒度太细；Pthreads（POSIX Thread）等为低端SMP标准
- 分布存储编程如MPI对程序员要求高，大量的科学应用程序需要很好地被继承和移植
- 通过使用OpenMP编程接口，可以较好的利用多核处理器的并发功能，提高程序的执行效率

Pthread/Java Thread /OpenMP

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

void* thread(void*)
{
    printf("Thread %d\n", (int)pthread_self());
    return NULL;
}

int main(int argc, char* argv[])
{
    pthread_t thread;
    int n_threads = 4;

    // set number of threads
    pthread_attr_t attr;
    pthread_attr_t attr;
    pthread_attr_t attr;

    // create threads
    for(i=0; i<n_threads; i++)
    {
        pthread_create(&thread, &attr, thread, (void*)i);
    }

    // wait for the N threads to finish
    for(i=0; i<n_threads; i++)
        pthread_join(thread, NULL);
}
```

```
class myth extends Thread
{
    myth (String name)
    {super(name); }

    public void run()
    {
        System.out.println("Thread " + name);
    }
}

class Hello
{
    public static void main(String[] args)
    {
        for(int i=0; i<args.length; i++)
        {
            new myth(args[i]).start();
        }
    }
}
```

```
#include <omp.h>
#include <stdio.h>
#define NUM_THREADS 4

int main(void)
{
    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    {
        printf("Hello from thread %d\n", omp_get_thread_num());
    }
}
```

OpenMP的目标

- 标准性 (**Standardization**)
 - 为共享体系结构 (平台) 提供编程标准
 - 多线程编程的高级接口
- 简洁实用 (**Lean and Mean**)
 - 3到4个编译制导已足够表达并行性
- 使用方便 (**Ease of use**)
 - 支持增量并行化
 - 支持粗粒度和细粒度的并行
- 可移植性 (**Portability**)
 - Fortran (77, 90, and 95), C/C++
 - API和成员的公共论坛 (www.openmp.org)

OpenMP不包含的性质

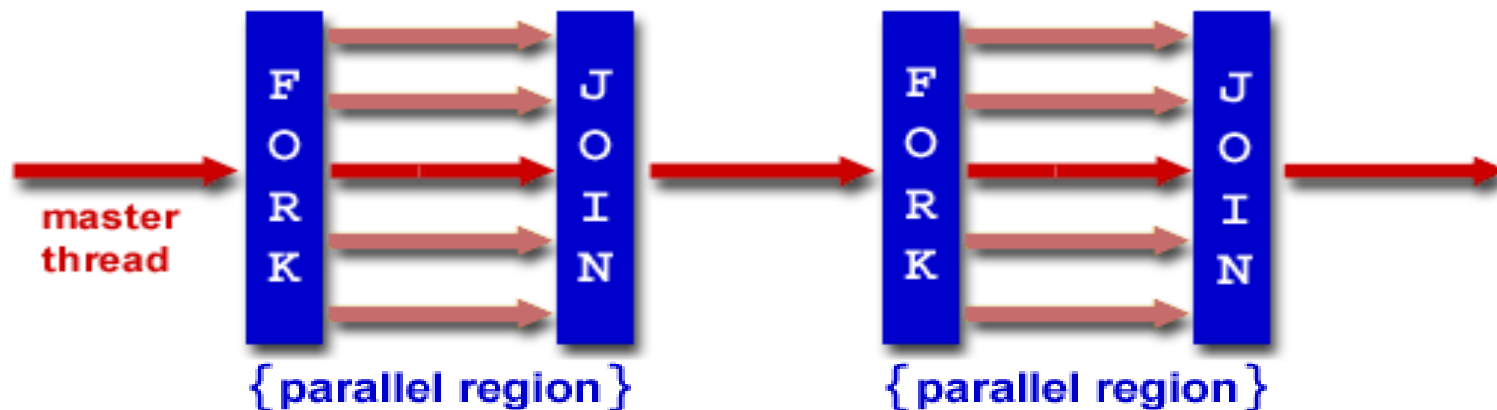
- 不是建立在分布式存储系统上的
- 不是在所有的环境下都是一样的
- 不是能保证让多数共享存储器均能有效的利用
- 已支持的编译器：
 - **Intel C/C++ and Fortran Compilers 10.0 (v2.5规范)**
 - **GCC 4.2(v2.5规范) , GCC 4.4 (v3.0规范)**
 - **Microsoft VC++ 2005, 2008 (v2.0规范)**
 - **Sun、IBM、HP、SGI、Fujitsu**

内容概要

- **OpenMP概述**
- **OpenMP编程模型**
- **OpenMP编程简介**
- **运行库例程与环境变量**
- **OpenMP计算实例**

OpenMP并行编程模型

- 基于**线程**的并行编程模型（**Programming Model**）
- OpenMP使用**Fork-Join**并行执行模型
 - **Fork**: 在并行结构的开头，**主线程**（**master thread**）创建了一队（**team**）的并行线程，并行域中的代码在不同的线程对中并行执行
 - **Join**: 在并行结构的末尾，所有线程同步或终止，只保留主线程



OpenMP编程模型

- 基于**线程**的并行（**Thread Based Parallelism**）
 - 一个有多个线程的共享存储进程
 - 显式并行：编程人员可以控制并行化
- 基于**编译制导**（**Compiler Directive Based**）
 - 并行化是通过嵌在C/C++或 Fortran源码中的编译制导来说明的
- 支持**嵌套**并行化（**Nested Parallelism Support**）
 - 并行区域可以嵌套其他的并行区域
- 支持**动态**线程（**Dynamic Threads**）
 - 可以改变并行区域中执行的线程数目

OpenMP程序结构

```
#include <omp.h>
main ()
{
    int var1, var2, var3;

    Serial code
    ...
    /* Beginning of parallel section. Fork a team of threads.
       Specify variable scoping */
    #pragma omp parallel private(var1, var2) shared(var3)
    {
        Parallel section executed by all threads
        ...
        All threads join master thread and disband
    }
    Resume serial code
}
```

一个简单的OpenMP程序实例

```
#include "omp.h"

int main(int argc, char* argv[])
{
    int nthreads, tid;
    int nprocs;
    char buf[32];
    /* Fork a team of threads */
    #pragma omp parallel private(nthreads, tid)
    {
        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from OMP thread %d\n", tid);
        /* Only master thread does this */
        if (tid==0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads %d\n", nthreads);
        }
    }
    return 0;
}
```

一个简单的OpenMP程序实例

- 运行结果 (setenv OMP_NUM_THREADS 8)

```
Hello World from OMP thread 0  
Number of threads 8  
Hello World from OMP thread 4  
Hello World from OMP thread 5  
Hello World from OMP thread 6  
Hello World from OMP thread 7  
Hello World from OMP thread 2  
Hello World from OMP thread 1  
Hello World from OMP thread 3
```

内容概要

- **OpenMP概述**
- **OpenMP编程模型**
- **OpenMP编程简介**
 - 控制结构
 - 数据域
- **运行库例程与环境变量**
- **OpenMP计算实例**

编译制导（Directive）语句格式

#pragma omp	Directive name	[clause, ...]	newline
制导指令前缀。 对所有的OpenMP语句都需要这样的前缀	OpenMP制导指令。在制导指令前缀和子句之间必须有一个正确的OpenMP制导指令	子句。在没有其它约束条件下，子句可以无序，也可以任意的选择。这一部分也可以没有	换行符，必需的。 表明这条制导语句的终止

例子： `#pragma omp parallel default(shared) private(beta, pi)`

编译制导作用域 (Scoping)

- 静态扩展 (Static extent) /词法扩展
 - 文本代码在一个编译制导语句之后，被封装到一个结构块中
- 孤立语句 (Orphan directives) /孤幼制导
 - 一个OpenMP的编译制导语句不依赖于其它的语句
- 动态扩展 (Dynamic extent)
 - 包括静态范围和孤立语句

编译制导作用域

<pre>#pragma omp parallel { ... #pragma omp for for(i=0; i<n; i++) { for(j=0; j<m; j++) sub1(); sub2(); } }</pre>	<pre>sub1() { #pragma omp critical ... } sub2() { #pragma omp sections ... }</pre>
Static extent	Orphan directives
Dynamic extent	

并行域结构 (Parallel Region)

- 并行域中的代码被所有的线程执行
- 具体格式
 - **#pragma omp parallel [clause[,]clause...]newline**
 - **clause=**
 - **if (scalar_expression)**
 - **private (list)**
 - **shared (list)**
 - **default (shared | none)**
 - **firstprivate (list)**
 - **reduction (operator: list)**
 - **copyin (list)**

Parallel编译制导

- 一遇到 **PARALLEL** 编译制导，一个线程创建一个团队的线程并成为主线程。主线程也是线程团队的一员。程序代码被复制，每个线程执行相同的代码。
- 线程团队的线程数目由下面决定（按次序）：
 - **omp_set_num_threads()** 库函数
 - **OMP_NUM_THREADS** 环境变量
- 线程的标识号从0 (主线程) 到 N-1。缺省地，程序的各个并行区域执行的线程数是相等的
- 动态线程：可以动态地调整线程的数目
 - **omp_set_dynamic()** 库函数
 - **OMP_DYNAMIC** 环境变量
- 在并行区域末尾有一个隐式的同步障。在同步障之后，只有主线程才继续执行

并行域结构的例子：计算 π

```
#include <omp.h>
static long num_steps = 100000000; double step;
#define NUM_THREADS 16

void main ()
{ int i; double x, pi, sum[NUM_THREADS];
  step = 1.0/(double) num_steps;
  omp_set_num_threads(NUM_THREADS);

  #pragma omp parallel
  { double x; int id; int i;
    id = omp_get_thread_num();

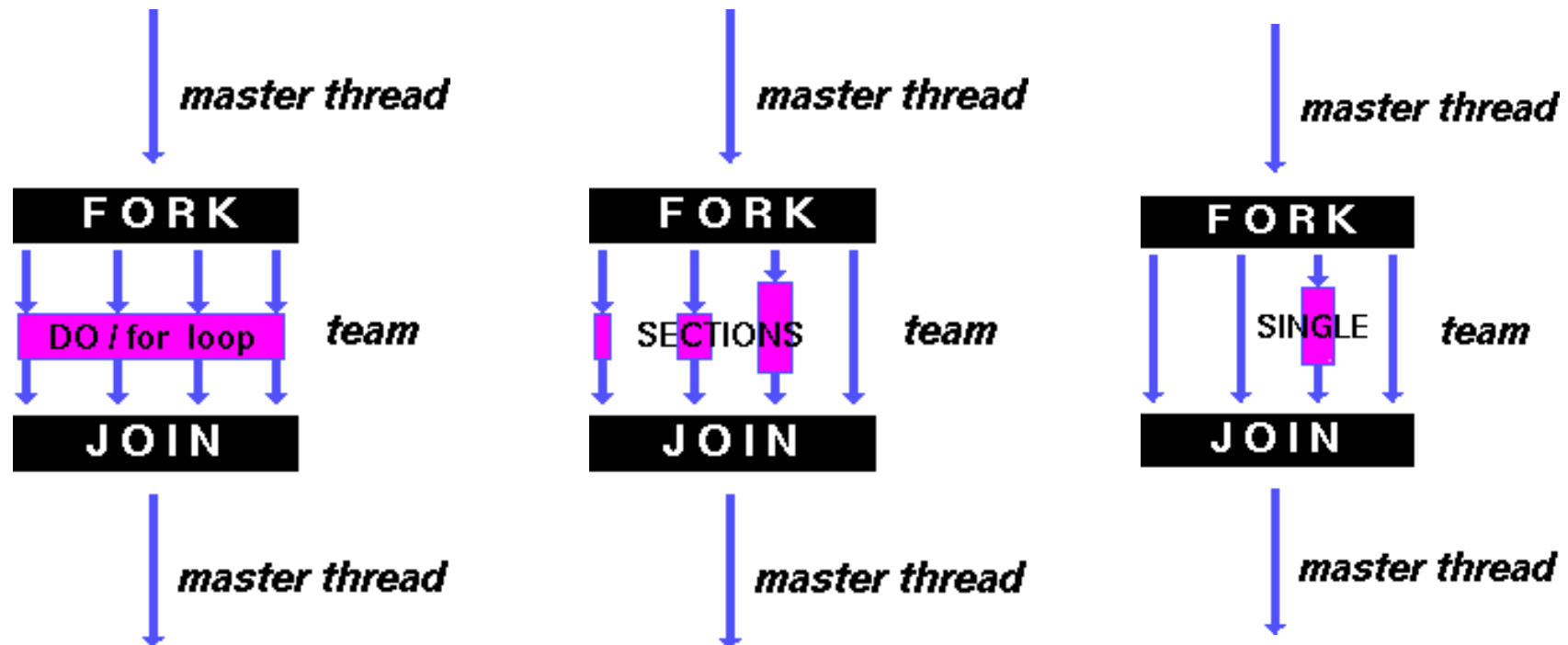
    printf("this is thread %d.\n",id);
    for (i=id, sum[id]=0.0; i<num_steps; i=i+NUM_THREADS){
      x = (i+0.5)*step;
      sum[id] += 4.0/(1.0+x*x);
    }
  }
  for(i=0, pi=0.0; i<NUM_THREADS; i++) pi += sum[i] * step;
  printf("Program succesfully terminated!\n");
  printf("PI is %lf.\n",pi);
}
```

```
scutgrid11{sbdong}6: pi1
this is thread 0.
this is thread 14.
this is thread 1.
this is thread 10.
this is thread 5.
this is thread 8.
this is thread 3.
this is thread 9.
this is thread 6.
this is thread 11.
this is thread 2.
this is thread 4.
this is thread 13.
this is thread 15.
this is thread 12.
this is thread 7.
Program succesfully terminated!
PI is 3.141593.
```

共享任务结构

(Work-Sharing Constructs)

- 共享任务结构将它所包含的代码划分给线程组的各成员来执行
 - 并行for循环
 - 并行sections
 - 串行执行

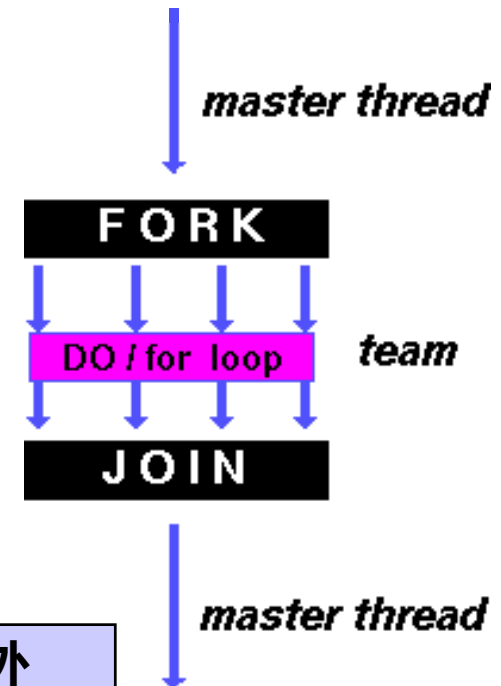


共享任务结构类型

- **#pragma omp for**
 - 将循环分布到在一个线程团队中
 - 可用来实现**数据并行**（**data parallelism**）
- **#pragma omp sections**
 - 将工作分成独立的，分散的部分
 - 每个部分由一个线程执行
 - 可用来实现**功能并行**（**functional parallelism**）
- **#pragma omp single**
 - **串行执行**

for编译制导语句

- **for**语句指定紧随它的循环语句必须由线程组并行执行;
- 语句格式
 - **#pragma omp for [clause[[,]clause]...] newline**
 - **[clause]=**
 - **Schedule(type [,chunk])**
 - **ordered**
 - **private (list)**
 - **firstprivate (list)**
 - **lastprivate (list)**
 - **shared (list)**
 - **reduction (operator: list)**
 - **nowait**



在for语句结束处有一个隐含的路障，使用了nowait子句除外

for编译制导语句

- **schedule**子句描述如何将循环的迭代划分给线程组中的线程
- 如果没有指定**chunk**大小，迭代会尽可能的平均分配给每个线程
- **type**为**static**，循环被分成大小为 **chunk**的块，静态分配给线程
- **type**为**dynamic**，循环被动态划分为大小为**chunk**的块，动态分配给线程
- **type**为**guided**，**chunk**的块大小指数递减
- 调度类型和**chunk**大小可以在运行时通过环境变量 **OMP_SCHEDULE**来设置

循环调度方式

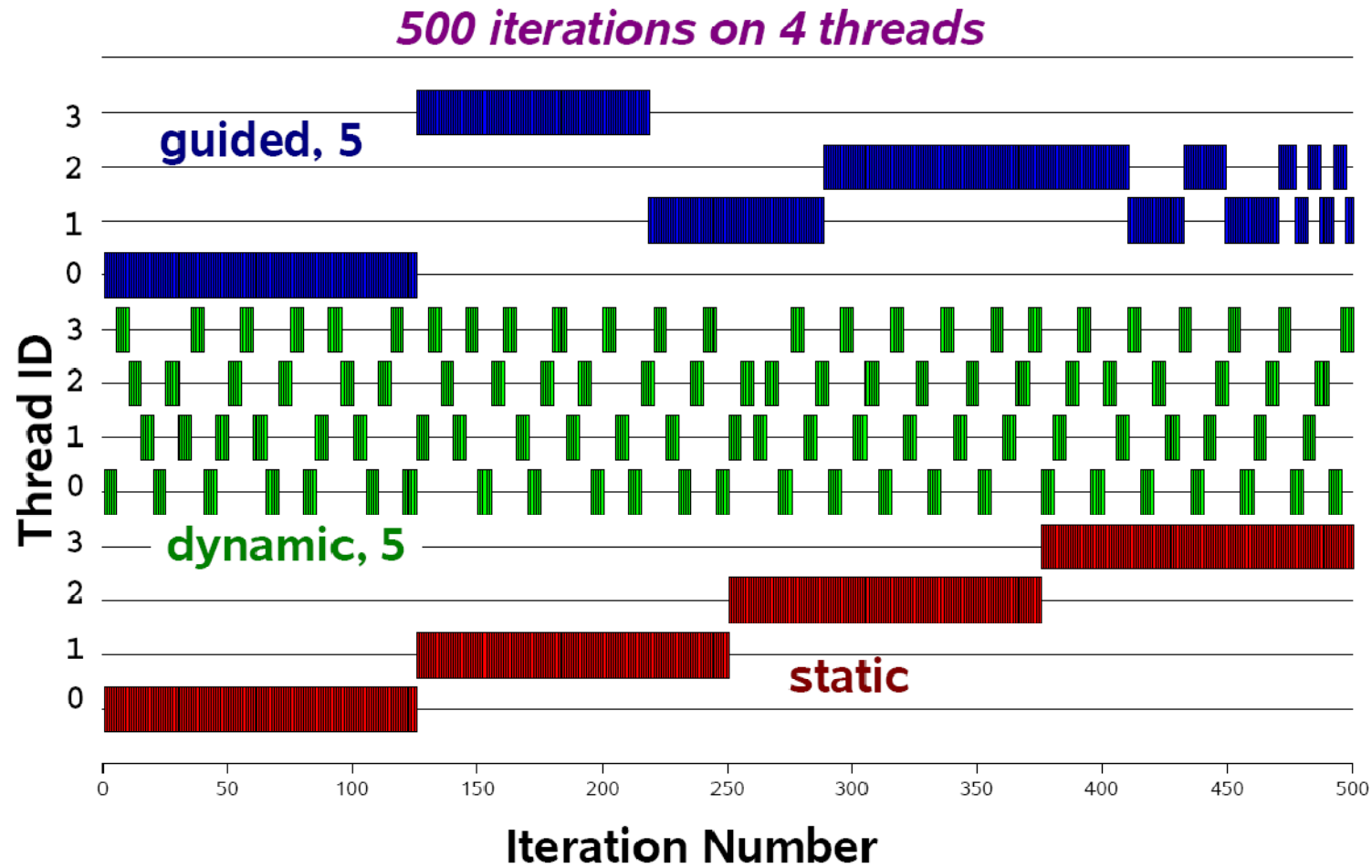
static

dynamic(3)

guided(1)



例子：不同模式下的调度实验



for编译制导语句的例子

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000

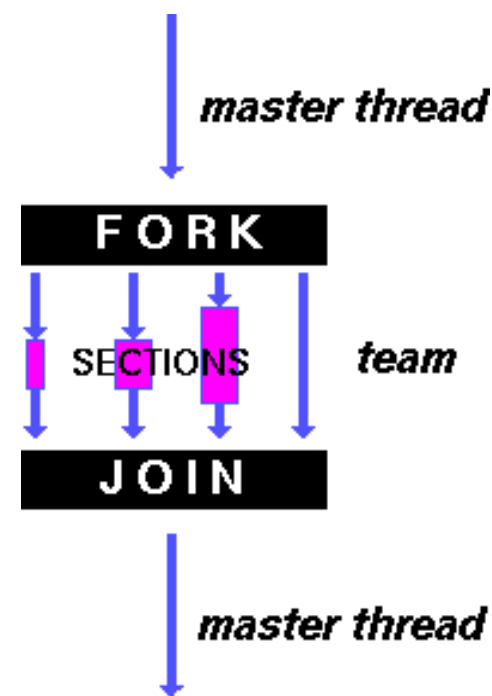
main ()
{
    int i, chunk;
    float a[N], b[N], c[N];

    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

    #pragma omp parallel shared(a,b,c,chunk) private(i)
    {
        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++)
            c[i] = a[i] + b[i];
    } /* end of parallel section */
}
```

Sections编译制导语句

- **sections**编译制导语句指定内部的代码被划分给线程组中的各线程
- 不同的**section**由不同的线程执行



Sections编译制导语句

- Section语句格式

#pragma omp sections [clause ...] newline

private (list)

firstprivate (list)

lastprivate (list)

reduction (operator: list)

nowait

{

#pragma omp section newline

structured_block

#pragma omp section newline

structured_block

}

在sections语句结束处有一个隐含的路障，使用了nowait子句除外

```

include <omp.h>
#define N 1000

main ()
{
    int i;
    float a[N], b[N], c[N];

    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    #pragma omp parallel shared(a,b,c) private(i)
    {
        #pragma omp sections nowait
        {
            #pragma omp section
            for (i=0; i < N/2; i++)
                c[i] = a[i] + b[i];

            #pragma omp section
            for (i=N/2; i < N; i++)
                c[i] = a[i] + b[i];
        } /* end of sections */
    } /* end of parallel section */
}

```

Section 编译 导语句的例子

single编译制导语句

- **single**编译制导语句指定内部代码只有线程组中的一个线程执行。用在多线程不安全的场合，如I/O
- 线程组中没有执行**single**语句的线程会一直等待代码块的结束，使用**nowait**子句除外
- 语句格式：

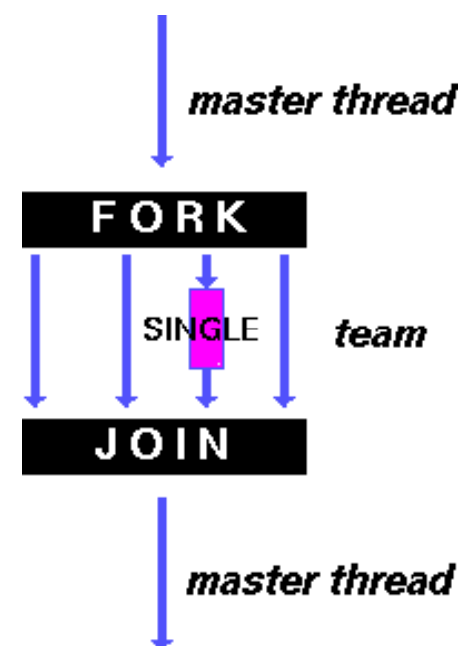
#pragma omp single [clause[[,]clause]...] newline

clause=

private(list)

firstprivate(list)

nowait



共享任务结构的例子：计算 π

```
#include <omp.h>
static long num_steps = 100000000; double step;
#define NUM_THREADS 16
void main ()
{
    int i; double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);

    #pragma omp parallel
    { double x; int id;
      id = omp_get_thread_num(); sum[id] = 0;
      printf("this is thread %d.\n",id);

      #pragma omp for
      for (i=id;i< num_steps; i++){
          x = (i+0.5)*step;
          sum[id] += 4.0/(1.0+x*x);
      }
    }
    for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
    printf("Program succesfully terminated!\n");
    printf("PI is %lf.\n",pi);
}
```

```
scutgrid11{sbdong}26: pi2
this is thread 0.
this is thread 6.
this is thread 1.
this is thread 2.
this is thread 10.
this is thread 3.
this is thread 14.
this is thread 12.
this is thread 5.
this is thread 7.
this is thread 8.
this is thread 9.
this is thread 4.
this is thread 11.
this is thread 13.
this is thread 15.
Program succesfully terminated!
PI is 3.141592.
```

组合的并行共享任务结构

- **parallel for**编译制导语句
 - **#pragma omp parallel + #pragma omp for**的简化
 - **Parallel for**编译制导语句表明一个并行域包含一个独立的for语句
- **parallel sections**编译制导语句
 - **#pragma omp parallel + #pragma omp sections**的简化
 - **parallel sections**编译制导语句表明一个并行域包含单独的一个sections语句

parallel for 编译制导语句的例子

```
#include <omp.h>
#define N    1000
#define CHUNKSIZE  100

main () {
    int i, chunk;
    float a[N], b[N], c[N];

    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i * 1.0;
    chunk = CHUNKSIZE;

    #pragma omp parallel for shared(a,b,c,chunk) private(i) schedule(static,chunk)
    for (i=0; i < n; i++)
        c[i] = a[i] + b[i];
}
```


OpenMP 同步

- 隐式同步
 - 在同步结构的开头和结尾
 - 任何控制结构的末尾
 - 隐式同步可以用**nowait**子句去除
- 显式同步
 - 同步结构（**synchronization constructs**）

同步结构

(Synchronization Constructs)

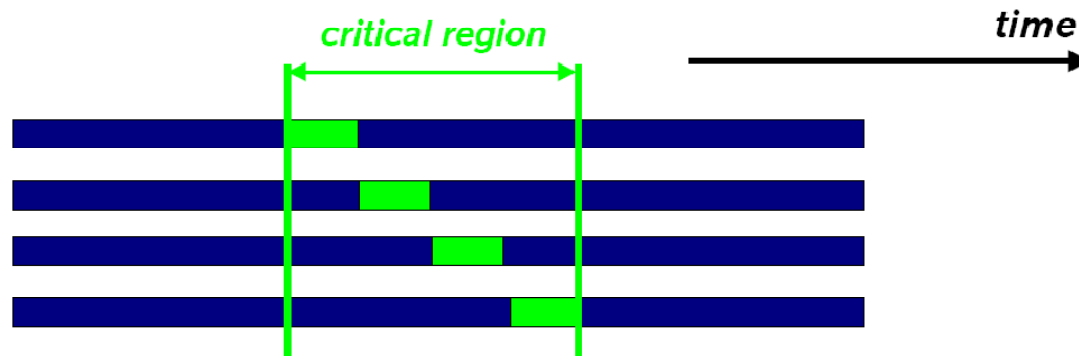
- **master** 制导语句
 - **#pragma omp master**
- **critical**制导语句
 - **#pragma omp critical**
- **barrier**制导语句
 - **#pragma omp barrier**
- **atomic**制导语句
 - **#pragma omp atomic**
- **flush**制导语句
 - **#pragma omp flush**
- **ordered**制导语句
 - **#pragma omp ordered**

master制导语句

- **master**制导语句指定代码段只有主线程执行
- 语句格式
 - **#pragma omp master newline**

critical制导语句

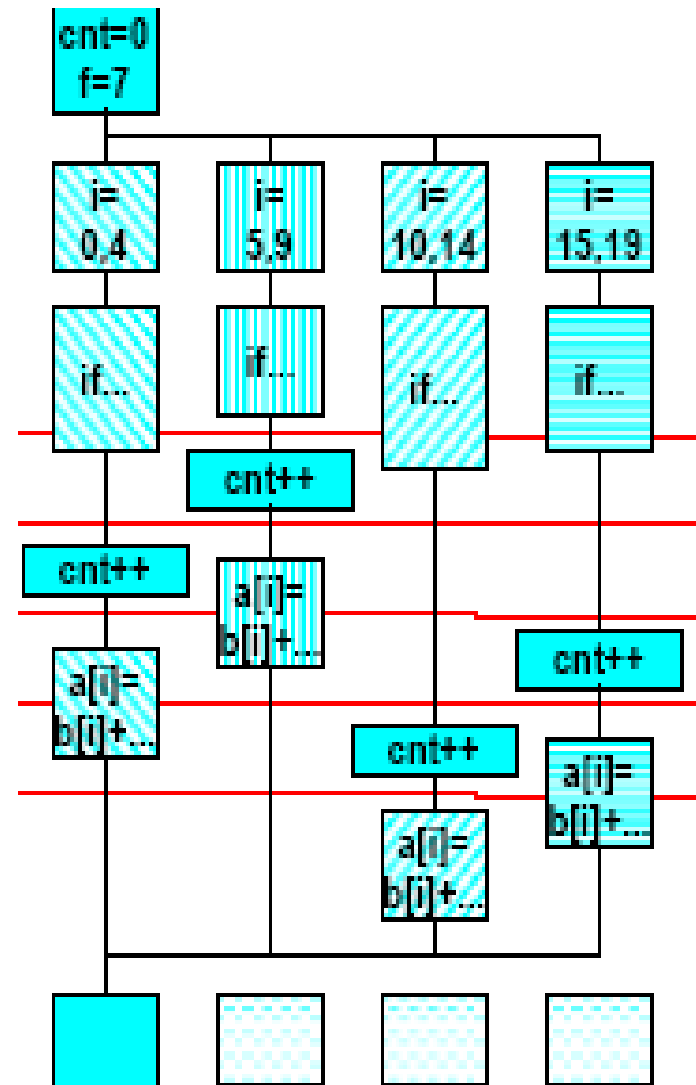
- **critical**制导语句表明域中的代码一次只能执行一个线程
- 其他线程被阻塞在临界区
- 语句格式:
 - `#pragma omp critical [name] newline`



critical 制导语句的例子

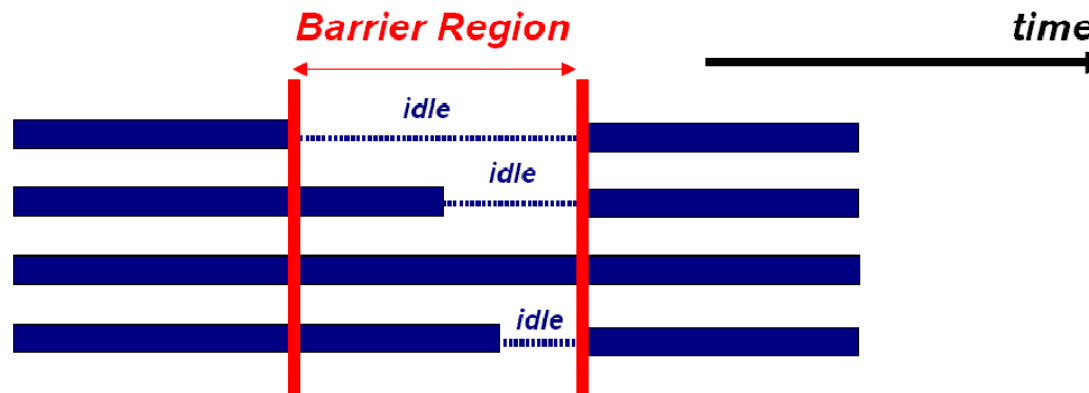
```
cnt = 0;
f=7;

#pragma omp parallel
{
  #pragma omp for
  for (i=0; i<20; i++) {
    if (b[i] == 0) {
      #pragma omp critical
      cnt ++;
    } /* endif */
    a[i] = b[i] + f * (i+1);
  } /* end for */
} /*omp end parallel */
```



barrier制导语句

- **barrier**制导语句用来同步一个线程组中所有的线程
- 先到达的线程在此阻塞，等待其他线程
- **barrier**语句最小代码必须是一个结构化的块
- 语句格式
 - `#pragma omp barrier newline`



例子：同步障

- 如果并行地执行以下两个语句，可能有什么情况出现？

```
for (i=0; i < N; i++)  
    a[i] = b[i] + c[i];
```

barrier

```
#pragma omp barrier
```

```
for (i=0; i < N; i++)  
    d[i] = a[i] + b[i];
```

This may give us a wrong answer (one day)

atomic制导语句

- **atomic**制导语句指定特定的存储单元将被原子更新
- 语句格式
 - **#pragma omp atomic newline**
- **atomic**使用的格式

```
x binop = expr  
x++  
++x  
x--  
--x
```

```
#pragma omp atomic  
a[indx[i]] += b[i];
```

x是一个标量

expr是一个不含对x引用的标量表达式，且不被重载

binop是+,*,-,/,&,<,>,or<<之一，且不被重载

flush 制导语句

- **flush**制导语句用以标识一个同步点，用以确保所有的线程看到一致的存储器视图
- 语句格式
 - **#pragma omp flush (list) newline**
- **flush**将在下面几种情形下隐含运行，**nowait**子句除外

barrier

critical:进入与退出部分

ordered:进入与退出部分

parallel:退出部分

for:退出部分

sections:退出部分

single:退出部分

ordered 制导语句

- **ordered** 制导语句指出其所包含循环的执行
- 任何时候只能有一个线程执行被 **ordered** 所限定部分
- 只能出现在 **for** 或者 **parallel for** 语句的动态范围中
- 语句格式:
 - **#pragma omp ordered newline**

总结：控制结构

- 并行域结构
 - **parallel**
- 共享工作结构
 - **sections, for, single**
- 组合的并行共享工作结构
 - **parallel for, parallel section**
- 同步结构
 - **master, critical, barrier, atomic, flush, ordered**

内容概要

- **OpenMP概述**
- **OpenMP编程模型**
- **OpenMP编程简介**
 - 控制结构
 - 数据域
- **运行库例程与环境变量**
- **OpenMP计算实例**

数据域

(Data Scope)

- 数据域属性子句 (**Data scope attribute clauses**) : 显式地定义变量的作用范围
 - **private**子句
 - **shared**子句
 - **default**子句
 - **firstprivate**子句
 - **lastprivate**子句
 - **copyin**子句
 - **reduction**子句
- **Data environment (数据环境)**
 - **threadprivate**编译制导语句

private和shared子句

- **private**子句表示它列出的变量对于每个线程是局部的
- 语句格式
 - **private(list)**
- **shared**子句表示它所列出的变量被线程组中所有的线程共享
- 所有线程都能对它进行读写访问
- 语句格式
 - **shared (list)**

default子句

- **default**子句让用户自行规定在一个并行域的静态范围中所定义的变量的缺省作用范围
- 语句格式
 - **default (shared | none)**

```
#pragma omp parallel default(none)\  
    shared(a,b,c) private(i)
```

firstprivate和lastprivate子句

- firstprivate子句是private子句的超集
- 对变量做原子初始化
- 语句格式:
 - firstprivate (list)
- lastprivate子句是private子句的超集
- 将变量从最后的循环迭代或段复制给原始的变量
- 语句格式
 - lastprivate (list)

firstprivate和lastprivate的例子

```
#include <omp.h>

main()
{
    int i,A,B,C;

    A = 10;
    #pragma omp parallel
    {
        #pragma omp for private(i) firstprivate(A) lastprivate (B)
        for (i=0; i<10; i++)
        {
            B = A + i;
        }
        C = B;
    }
    printf("A=%d,B=%d,C=%d\n",A,B,C);
}
```

```
scutgrid11{sbdong}2: ./first
A=10,B=19,C=19
```

threadprivate编译制导语句

- **threadprivate**语句使一个全局文件作用域的变量在并行域内变成每个线程私有
- 每个线程对该变量复制一份私有拷贝
- 语句格式:
 - **#pragma omp threadprivate (list) newline**

private和threadprivate区别

	PRIVATE	THREADPRIVATE
数据类型	变量	变量
位置	在域的开始或共享任务单元	在块或整个文件区域的例程的定义上
持久性	否	是
扩充性	只是词法的- 除非有个子程序	动态的
初始化	使用 FIRSTPRIVATE	使用 COPYIN

threadprivate编译制导语句的例子

```
int alpha[10], beta[10], i;  
#pragma omp threadprivate(alpha)
```

```
main ()
```

```
{
```

```
    /* First parallel region */
```

```
    #pragma omp parallel private(i,beta)
```

```
    for (i=0; i < 10; i++)
```

```
        alpha[i] = beta[i] = i;
```

```
    /* Second parallel region */
```

```
    #pragma omp parallel
```

```
    printf("alpha[3]= %d and beta[3]= %d\n",alpha[3],beta[3]);
```

```
}
```

```
scutgrid11{sbdong}48: more data1.c  
int alpha[10], beta[10], i;  
#pragma omp threadprivate(alpha)  
  
main ()  
{  
    /* First parallel region */  
    #pragma omp parallel private(i,beta)  
    for (i=0; i < 10; i++)  
        alpha[i] = beta[i] = i;  
  
    /* Second parallel region */  
    #pragma omp parallel  
    printf("alpha[3]= %d and beta[3]= %d\n",alpha[3],beta[3]);  
}  
  
scutgrid11{sbdong}49: data1  
alpha[3]= 3 and beta[3]= 0
```

copyin子句

- **copyin**子句用来为线程组中所有线程的**threadprivate**变量赋相同的值
- 主线程该变量的值作为初始值
- 语句格式
 - **copyin(list)**

```
#pragma omp parallel copyin(a,b,c)
```

reduction子句

- **reduction**子句使用指定的操作对其列表中出现
的变量进行规约
- 初始时，每个线程都保留一份私有拷贝
- 在结构尾部根据指定的操作对线程中的相应变
量进行规约，并更新改变量的全局值
- 语句格式
 - **reduction (operator: list)**

reduction子句

- Reduction子句的格式

```
x=x op expr
x = expr op x (except subtraction)
x binop = expr
x++
++x
x--
--x
```

x是一个标量

expr是一个不含对x引用的标量表达式，且不被重载

binop是+,*,-,/,&^,|之一，且不被重载

op是+,*,-,/,&^,|,&&,or之一，且不被重载

Reduction的例子

```
#include <omp.h>

main () {
    int i, n, chunk;
    float a[100], b[100], result;

    /* Some initializations */
    n = 100;
    chunk = 10;
    result = 0.0;
    for (i=0; i < n; i++) {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
    }

    #pragma omp parallel for default(shared) private(i) schedule(static,chunk) \
        reduction(+:result)
    for (i=0; i < n; i++)
        result = result + (a[i] * b[i]);
    printf("Final result= %f\n",result);
}
```


子句/编译制导语句总结

子句	编译制导					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	√				√	√
PRIVATE	√	√	√	√	√	√
SHARED	√	√			√	√
DEFAULT	√				√	√
FIRSTPRIVATE	√	√	√	√	√	√
LASTPRIVATE		√	√		√	√
REDUCTION	√	√	√		√	√
COPYIN	√				√	√
SCHEDULE		√			√	
ORDERED		√			√	
NOWAIT		√	√	√		

重要概念

- **Parallel region (并行域)**
- **Work-Sharing Constructs (共享任务结构)**
 - **#pragma omp for**
 - **#pragma omp single**
 - **#pragma omp sections**
- **Combined Parallel Work-Sharing Constructs (组合的共享任务结构)**
 - **#pragma omp parallel for**
 - **#pragma omp parallel sections**
- **Synchronization Constructs (同步结构)**
- **Data environment (数据环境)**
 - **#pragma omp threadprivate**
- **Data Scope Clauses (数据域属性子句)**

内容概要

- **OpenMP概述**
- **OpenMP编程模型**
- **OpenMP编程简介**
- **运行库例程与环境变量**
- **OpenMP计算实例**

运行库例程

- 运行库例程
 - **OpenMP**标准定义了一个应用编程接口来调用库中的多种函数
 - 对于**C/C++**，在程序开头需要引用文件“**omp.h**”

环境变量

- **OMP_SCHEDULE**
 - 只能用到for,parallel for中。它的值就是处理器中循环的次数
 - `setenv OMP_SCHEDULE "guided, 4"`
 - `setenv OMP_SCHEDULE "dynamic"`
- **OMP_NUM_THREADS**
 - 定义执行中最大的线程数
 - `setenv OMP_NUM_THREADS 8`
- **OMP_DYNAMIC**
 - 通过设定变量值TRUE或FALSE，来确定是否动态设定并行域执行的线程数
 - `setenv OMP_DYNAMIC TRUE`
- **OMP_NESTED**
 - 确定是否可以并行嵌套
 - `setenv OMP_NESTED TRUE`

内容概要

- **OpenMP概述**
- **OpenMP编程模型**
- **OpenMP编程简介**
- **运行库例程与环境变量**
- **OpenMP计算实例**

串行程序

```
/* Serial Code */
static long num_steps = 100000;
double step;
void main ()
{
    int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

使用并行域并行化

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
    int i;
    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS)
    #pragma omp parallel
    {
        double x;
        int id;
        id = omp_get_thread_num();
        for (i=id, sum[id]=0.0;i< num_steps;
i=i+NUM_THREADS){
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```


使用共享任务结构并行化

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
    int i;
    double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS)
    #pragma omp parallel
    {
        double x;
        int id;
        id = omp_get_thread_num();
        sum[id] = 0;
        #pragma omp for
        for (i=id;i< num_steps; i++){
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

使用private和critical并行化

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{
    int i;
    double x, sum, pi=0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS)
    #pragma omp parallel private (x, sum)
    {
        id = omp_get_thread_num();
        for (i=id,sum=0.0;i< num_steps;i=i+NUM_THREADS){
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
        #pragma omp critical
        pi += sum
    }
}
```

使用并行归约

```
#include <omp.h>
static long num_steps = 100000;
double step;
#define NUM_THREADS 2
void main ()
{   int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS)
    #pragma omp parallel for reduction(+:sum) private(x)
    for (i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

课程小结

- **Java thread**
 - 创建线程 (Threading Mechanisms)
 - 线程周期 (Thread cycle)
 - 资源共享、同步 (Synchronization)
 - 优先级 (Priority)
- **OpenMP**
 - 编程模型: **fork-join model**
 - 组成部分: **Directives, Runtime libraries, Environment variables**
 - 控制结构
 - **Parallel region**
 - **Work-Sharing Constructs**
 - **Synchronization Constructs**
 - 数据域子句 (**Data Scope Clauses**)

OpenMP vs. 线程

- **OpenMP优势:**
 - 工业标准，可移植性。
 - 对大数据量for循环计算的任务并行非常简单方便
- **OpenMP劣势:**
 - 数据量必须足够大，否则对于较小的数据集线程调度的开销反而会使性能下降
 - 并不适应所有类型的系统，比较适合于科学计算
 - 相比一开始就设计好的多线程系统来说，OpenMP的并行粒度和灵活性都较之要低

第二次作业

- 《并行计算—结构、算法、编程》
 - 5.10
 - 6.3
 - 7.3
 - 9.9
 - 12.5
 - 14.3

推荐网站和读物

- 《并行计算》
 - 第13章：共享存储系统并行编程
- **OpenMP | OpenMP: Simple, Portable, Scalable SMP Programming**
 - <http://www.openmp.org>
- **OpenMP Specifications Version 2.5**
- **OpenMP in Visual C++**
 - <http://msdn.microsoft.com/en-us/library/tt15eb9t.aspx>
- **Introduction to OpenMP - tutorial from Livermore Computing (LC)**
 - <http://www.llnl.gov/computing/tutorials/openMP/>

下一讲

- 消息传递编程
 - 《并行计算—结构、算法、编程》第14章