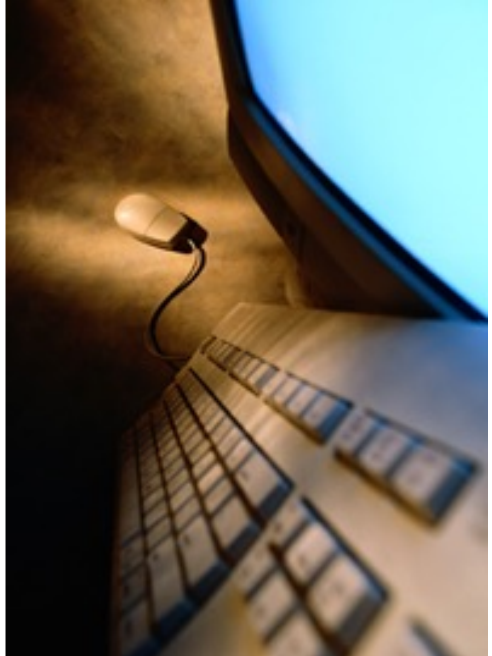
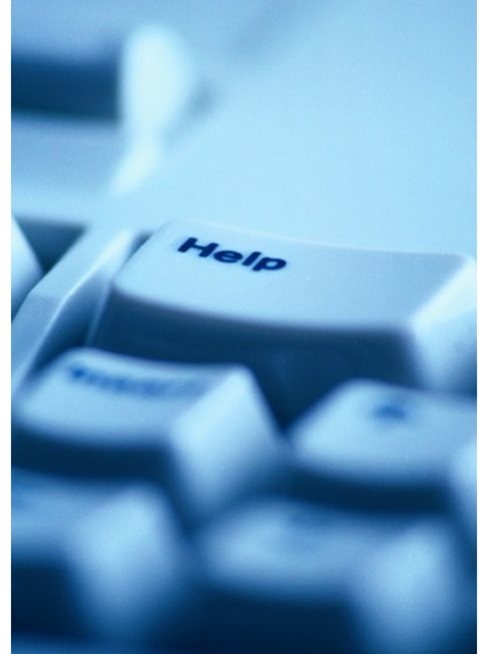
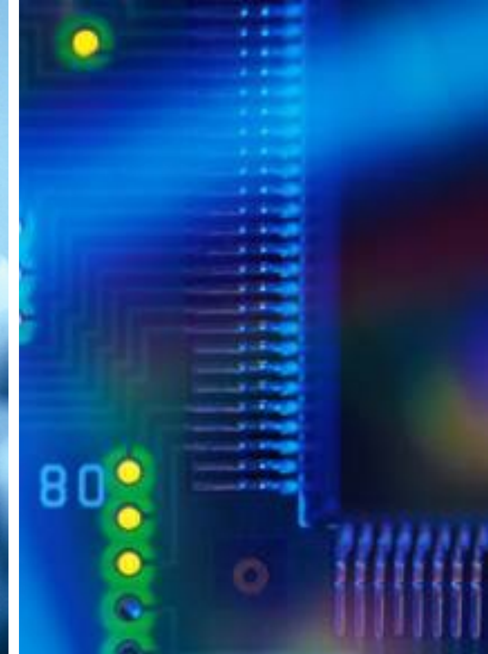






并行与分布式计算

Parallel & Distributed Computing

陈鹏飞

计算机学院

Lecture 6 — Race Conditions and Synchronization in OpenMP

Pengfei Chen

School of Data and Computer Science

Outline:

- **Correctness issues in parallel programming (in OpenMP)**
  - **Barriers （障碍，屏障）**
  - **Examples of race conditions**
  - **Mutual Exclusion （互斥）**
  - **Memory fence （内存屏障）**

Concept: Synchronization

- **Synchronization**
  - **The process of managing shared resources so that reads and writes occur in the correct order regardless of how the threads are scheduled**
- **Synchronization methods**
  - **Barriers**
  - **Mutual Exclusion （互斥） e.g.** pthread_mutex_lock
  - **…**

Barriers in OpenMP

- **Barrier**
  - **A synchronization point at which every member in a team of threads must arrive before any member can proceed**
- **Syntax**

  #pragma omp barrier

  - **Automatically inserted at the end of worksharing constructs**
  - **e.g.,** *for* **pragma,** *single* **pragma, …**
  - **Can be disabled by using the** *nowait* **clause**

Example: Use of Barrier

```
int numt = omp_get_num_threads();
#pragma omp parallel shared(numt)
{
    int tid = omp_get_thread_num();
    printf("hi, from %d\n", tid);
    if (tid == 0) {
        printf("%d threads say hi!\n", numt);
    }
}
```

Output using 4 threads

```
hi, from 3
hi, from 0
hi, from 2
4 threads say hi!
hi, from 1
```

- **Question:**
  - **What's the expected output?**
  - **How can we let the last printf appear last?**

Barbara Chapman, "A Guide to OpenMP," 2010.

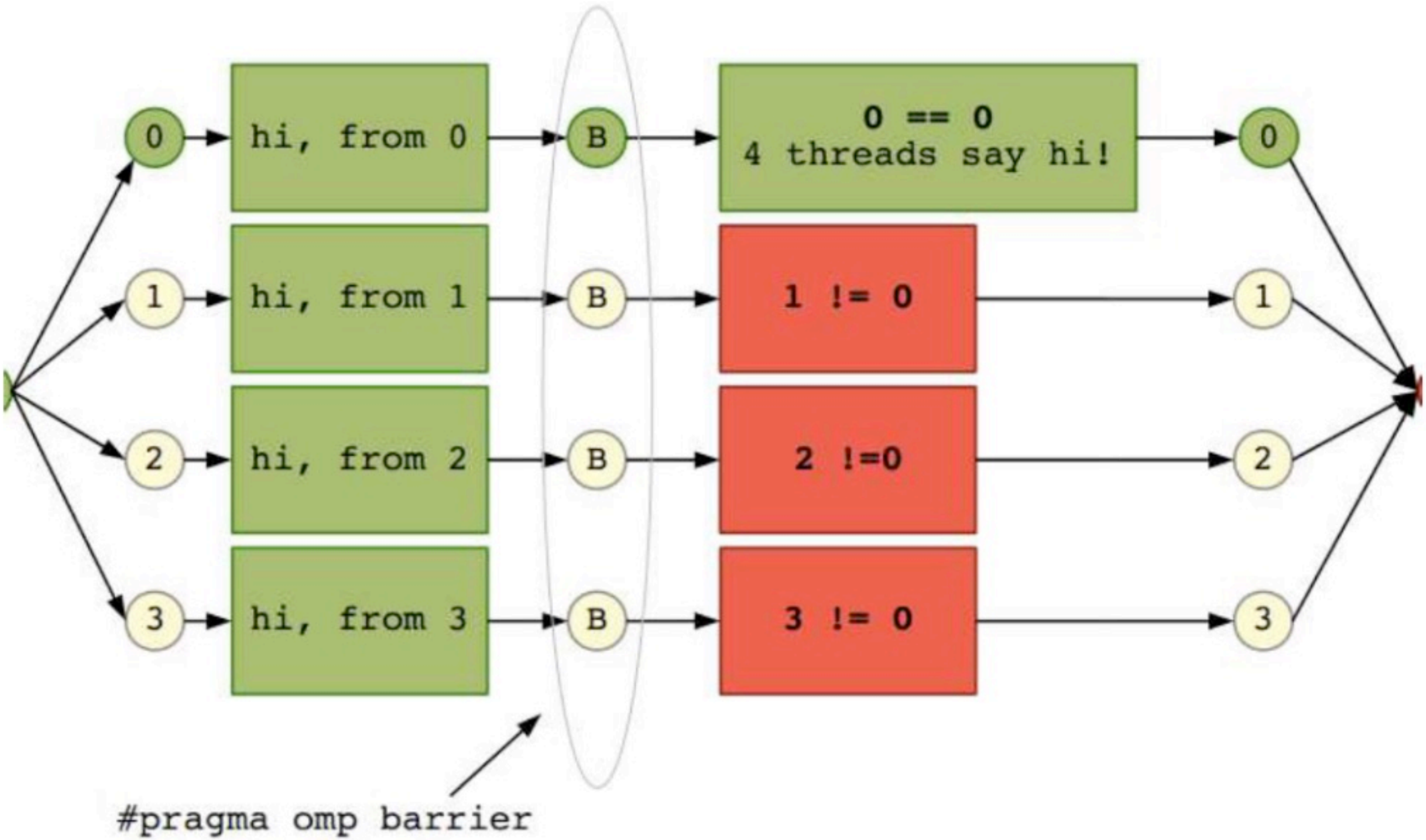
Example: an Explicit Barrier

```
int numt = omp_get_num_threads();
#pragma omp parallel shared(numt)
{
    int tid = omp_get_thread_num();
    printf("hi, from %d\n", tid);
#pragma omp barrier
    if (tid == 0) {
        printf("%d threads say hi!\n", numt);
    }
}
```

Output using 4 threads

hi, from 3
hi, from 0
hi, from 2
hi, from 1
4 threads say hi!

Example: an Explicit Barrier



Clause: *nowait*

- **The *nowait* clause tells the compiler that there is no need for a barrier synchronization at the end of a parallel *for* loop or *single* block of code**

Case: *parallel, for, single* ***Pragmas***

```
for (i = 0; i < N; i++)
      a[i] = alpha(i);
if (delta < 0.0)
      printf("delta < 0.0\n");
for (i = 0; i < N; i++)
     b[i] = beta(i, delta);
```

Solution: *parallel, for, single* **Pragmas**

```
#pragma omp parallel
{
    #pragma omp for nowait
    for (i = 0; i < N; i++)
        a[i] = alpha(i);
    #pragma omp single nowait
    if (delta < 0.0)
        printf("delta < 0.0\n");
    #pragma omp for
    for (i = 0; i < N; i++)
        b[i] = beta(i, delta);
}
```

← single ,同一时间只有一个线程执行

nowait ,执行完不等待

Mutual Exclusion

- Mutual exclusion
  - A kind of synchronization
  - Allows only a single thread or process at a time to have access to shared resource
  - Implemented using some form of locking
- Critical section (a high-level synchronization)
  - Only one thread at a time will execute the structured block within a critical section
- Lock (a low-level synchronization)

An Example of Race Condition

```
double area, pni, x;
int i, n;
...
area = 0.0;
for (i = 0; i <n; i++) {
   x = (i + 0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

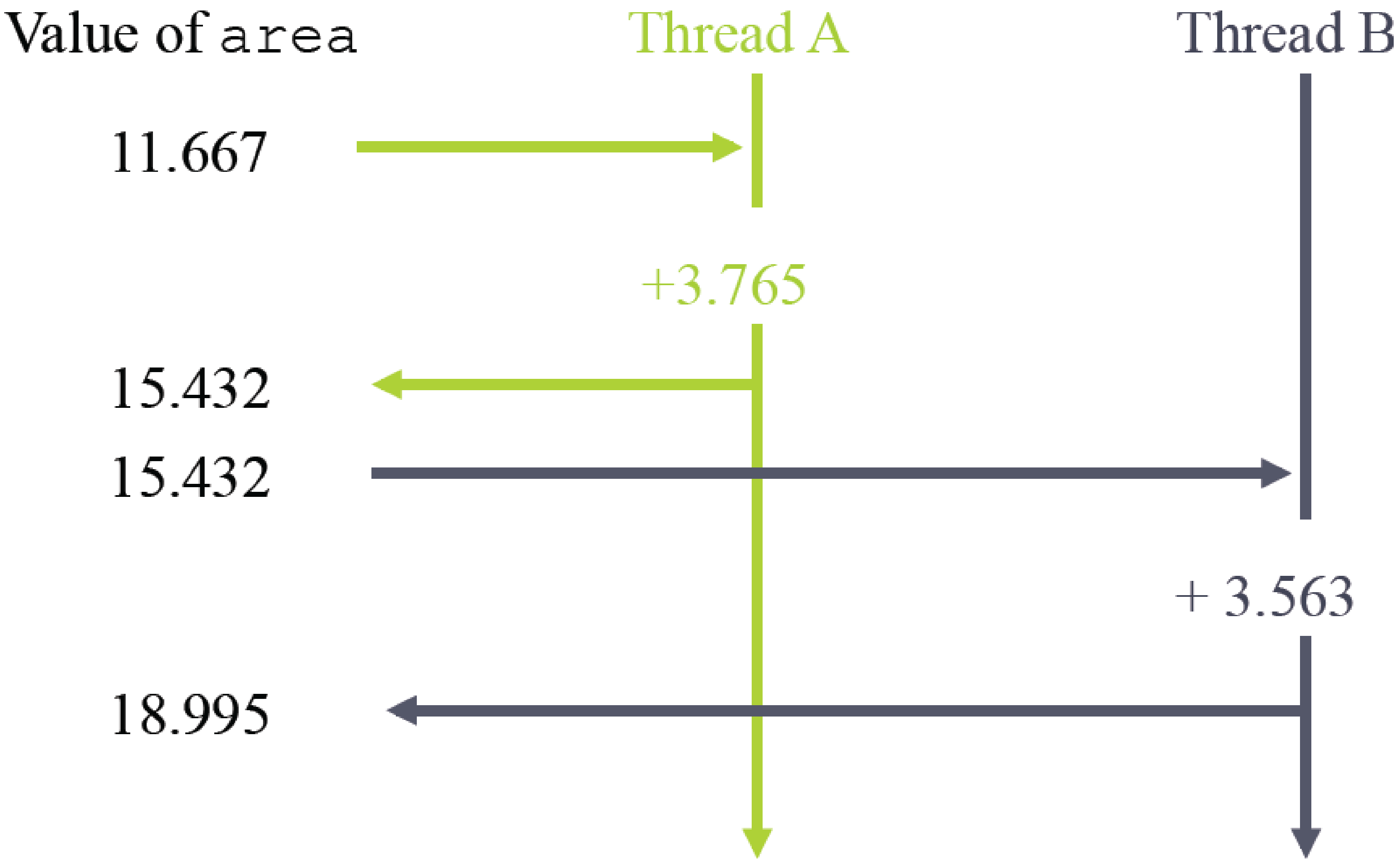
- **What happens when we make the `for` loop parallel?**

Race Condition

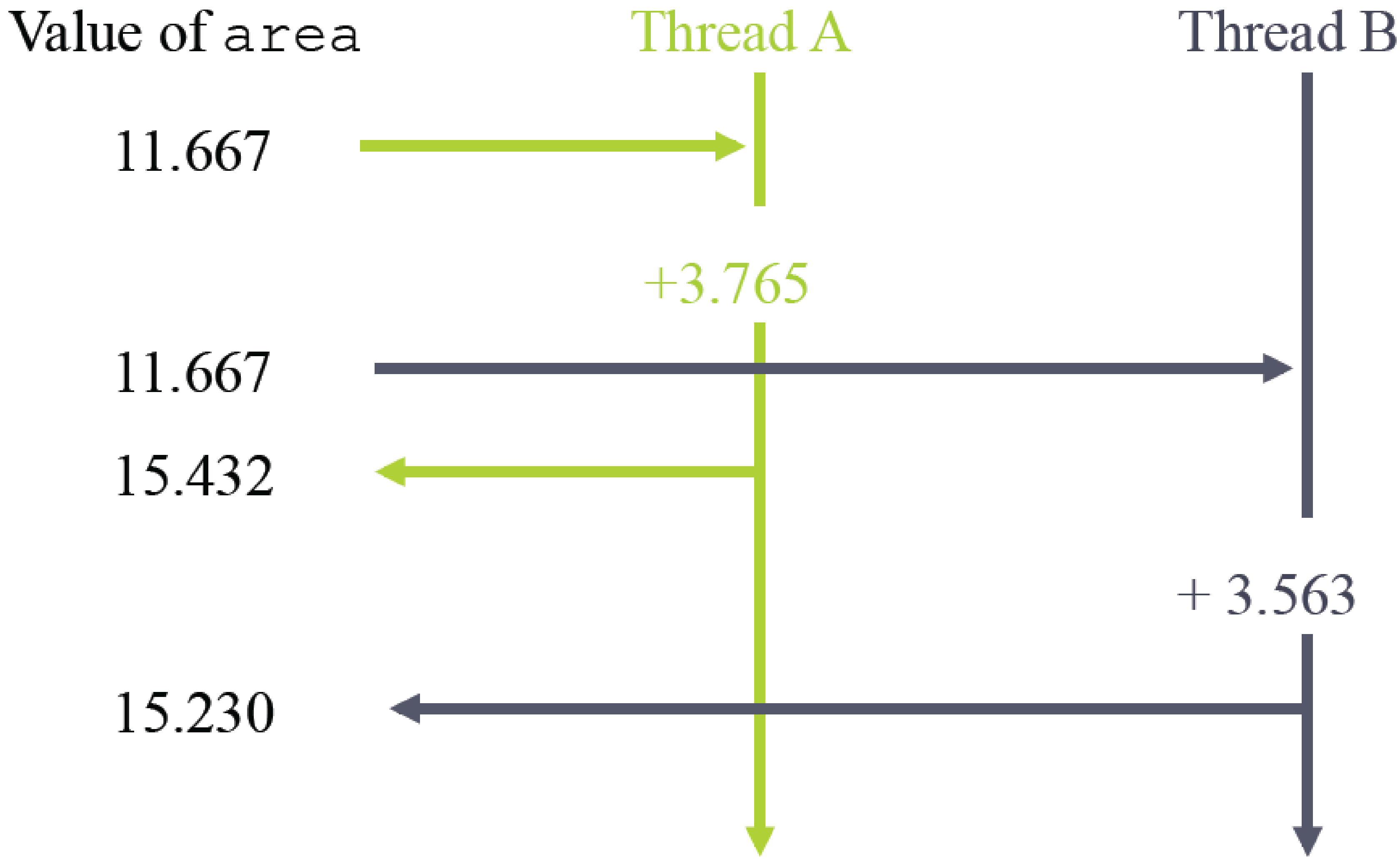
- **A *race condition* is nondeterministic （非确定性）behavior caused by the times at which two or more threads access a shared variable**
- **For example, suppose both Thread A and Thread B are executing the statement…**

```
area += 4.0 / (1.0 + x*x);
```

One Timing ⇒ Correct Sum



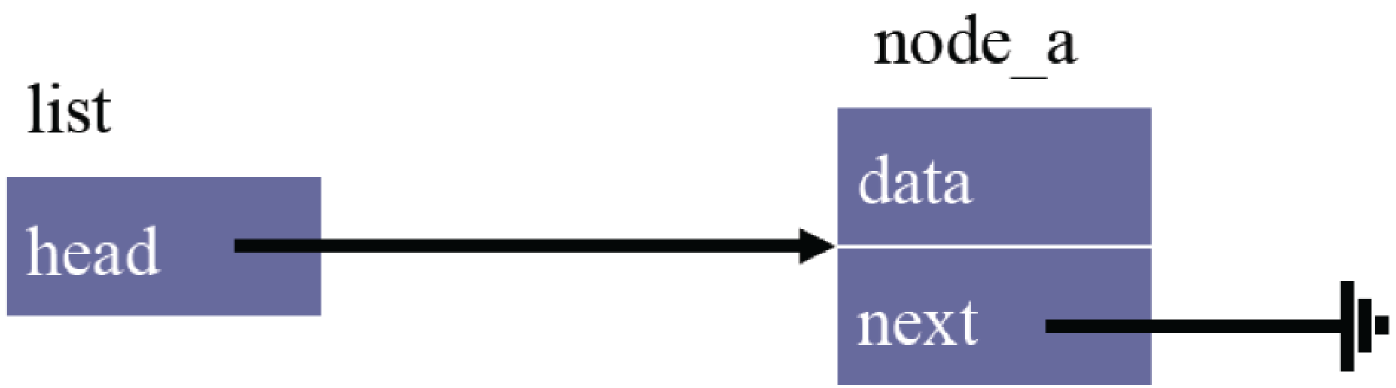
Another Timing ⇒ Incorrect Sum



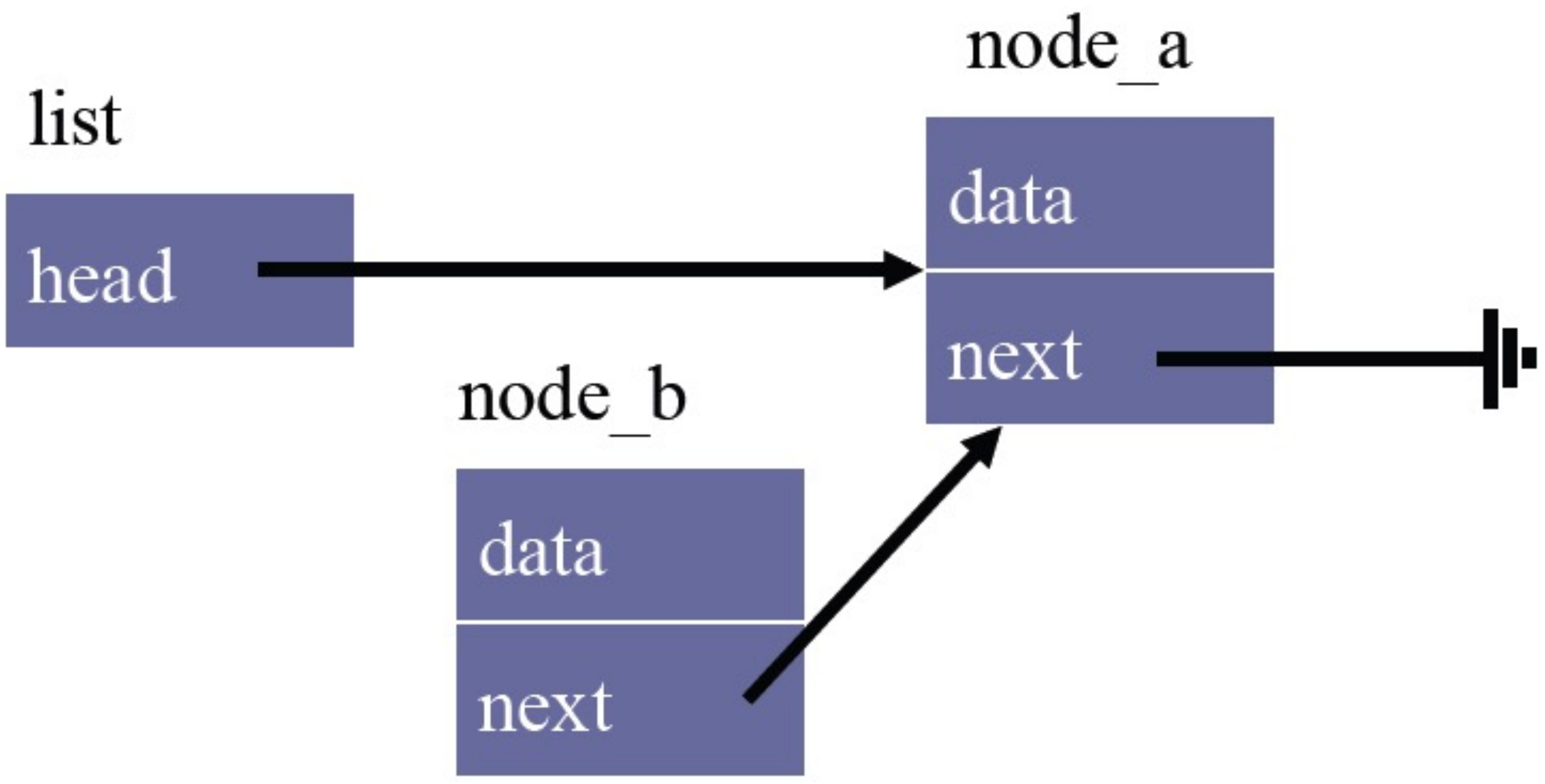
Another Race Condition Example

```
struct Node {
        struct Node* next;
         int data;
}
struct List {
         struct Node* head;
};
void AddHead (struct List* list, struct Node* node) {
          node->next = list->head;
          list->head = node;
}
```

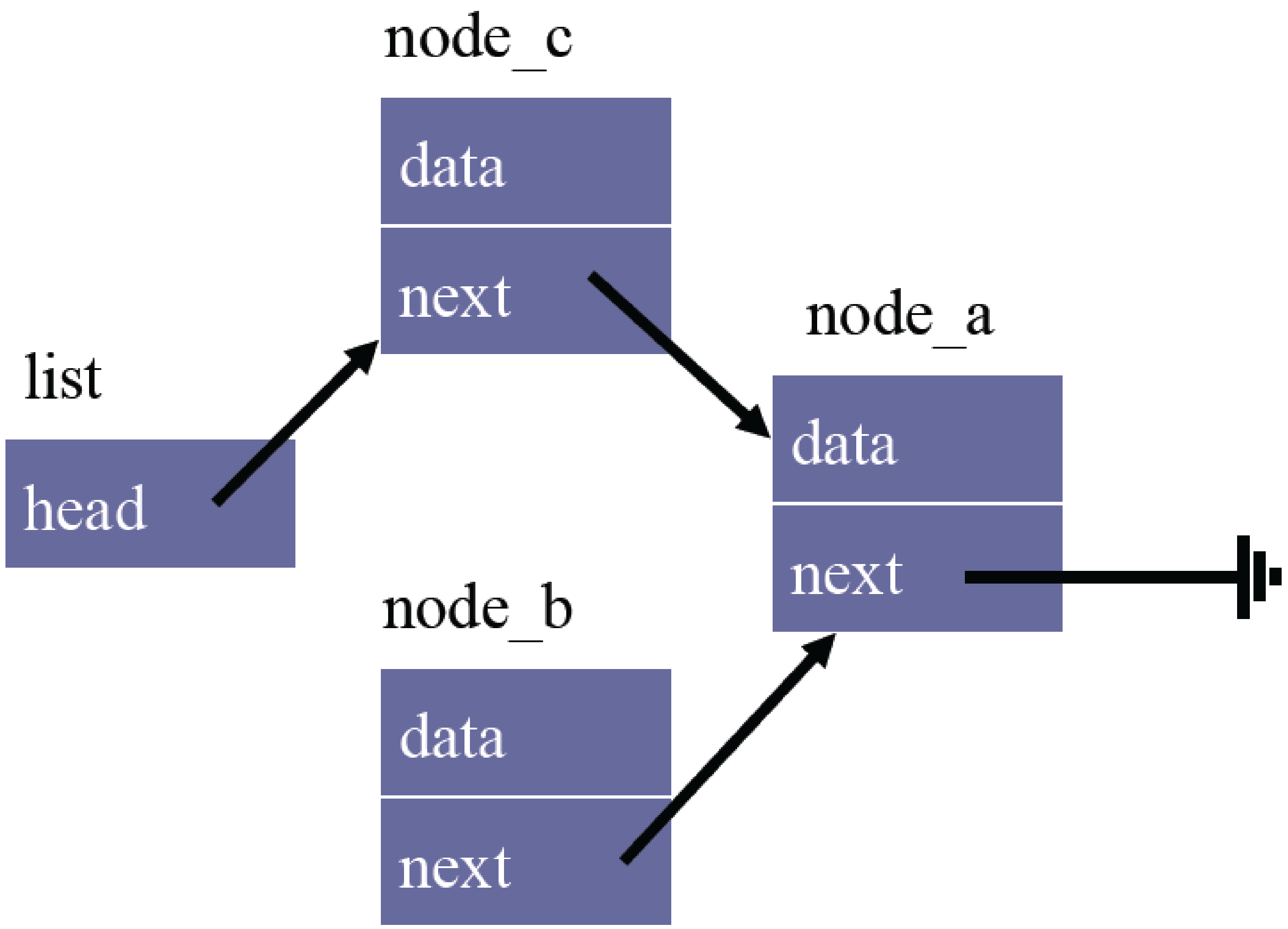
Original Singly-Linked List



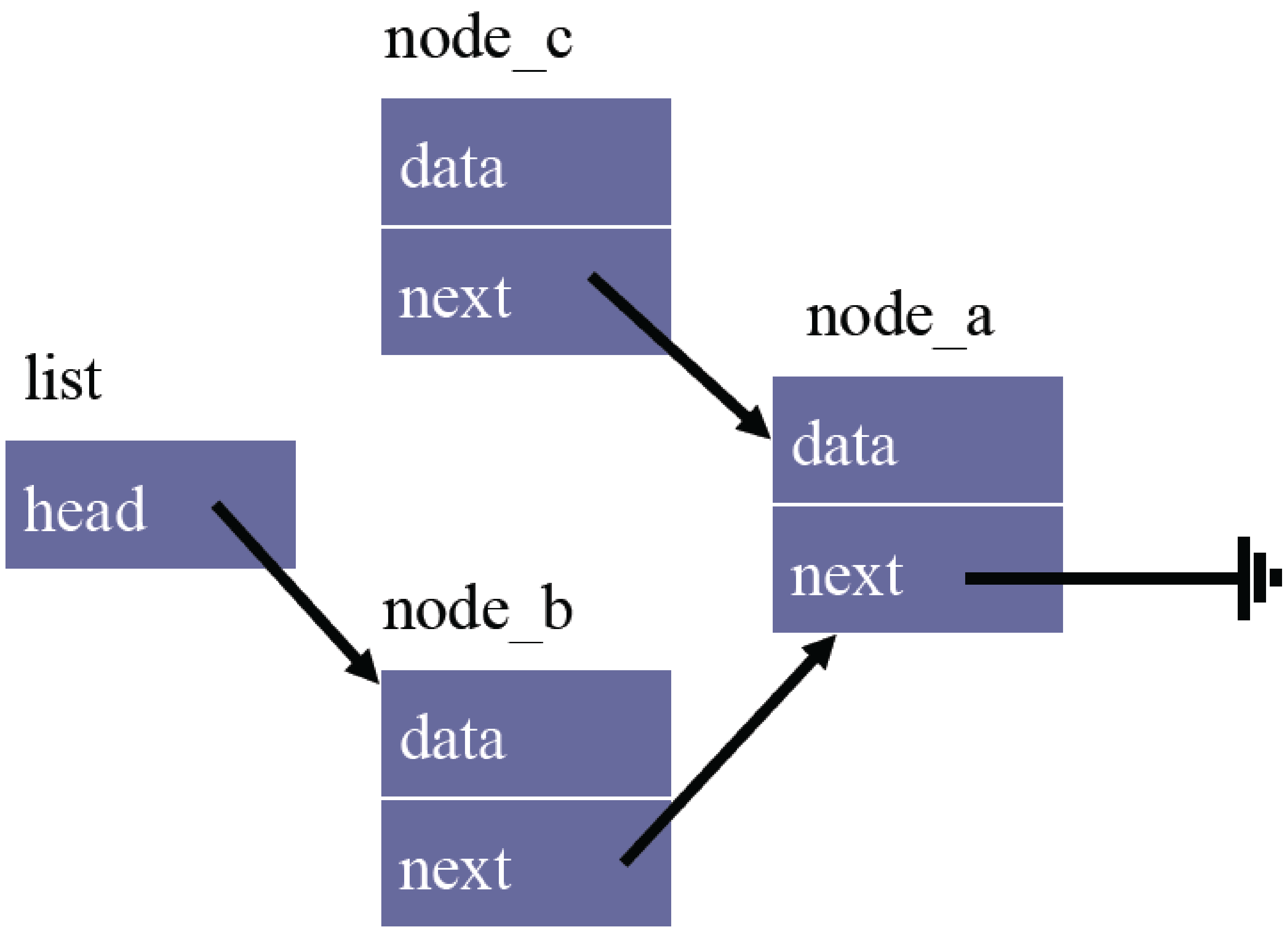
Thread 1 after Stmt. 1 of AddHead



Thread 2 Executes AddHead



Thread 1 after Stmt. 2 of AddHead



Why Race Conditions Are Nasty（令人讨厌）

- Programs with race conditions exhibit nondeterministic behavior
  - Sometimes give correct result
  - Sometimes give erroneous result
- Programs often work correctly on trivial data sets and small number of threads
- Errors more likely to occur when number of threads and/or execution time increases
- Hence debugging race conditions can be difficult

How to Avoid Race Conditions

- **Scope variables to be private to threads**
  - **Use OpenMP *private* clause**
  - **Variables declared within threaded functions**
  - **Allocate on thread's stack (pass as parameter)**
- **Control shared access with critical region**
  - **Mutual exclusion and synchronization**

Mutual Exclusion

- **We can prevent the race conditions described earlier …**
  - **Ensure that only one thread at a time references or updates shared variables**
- **Mutual exclusion**
  - **A kind of synchronization**
  - **Allows only a single thread or process at a time to have access to shared resource**
  - **Implemented using some form of locking**

Do Flags Guarantee Mutual Exclusion?

```
int flag = 0;

void AddHead(struct List* list, struct Node* node) {
    while (flag != 0) /* wait */;
    flag = 1;
    node->next = list->head;
    list->head = node;
    flag = 0;
}
```

Flags Don't Guarantee Mutual Exclusion

flag

0

Thread 1

```
int flag = 0;

void AddHead(struct List* list, struct Node* node) {
    while (flag != 0) /* wait */;
    flag = 1;
    node->next = list->head;
    list->head = node;
    flag = 0;
}
```

Flags Don't Guarantee Mutual Exclusion

flag

$\boxed{0}$

Thread 1   Thread 2

```
int flag = 0;

void AddHead(struct List* list, struct Node* node) {
    while (flag != 0) /* wait */;
    flag = 1;
    node->next = list->head;
    list->head = node;
    flag = 0;
}
```

Flags Don’t Guarantee Mutual Exclusion

flag

1

Thread 1 Thread 2

```
int flag = 0;

void AddHead(struct List* list, struct Node* node) {
    while (flag != 0) /* wait */;
    flag = 1;
    node->next = list->head;
    list->head = node;
    flag = 0;
}
```

Flags Don’t Guarantee Mutual Exclusion

flag

$\boxed{1}$

Thread 1 Thread 2

```
int flag = 0;

void AddHead(struct List* list, struct Node* node) {
    while (flag != 0) /* wait */;
    flag = 1;
    node->next = list->head;
    list->head = node;
    flag = 0;
}
```

Flags Don't Guarantee Mutual Exclusion

flag

0

Thread 1 Thread 2

```
int flag = 0;

void AddHead(struct List* list, struct Node* node) {
    while (flag != 0) /* wait */;
    flag = 1;
    node->next = list->head;
    list->head = node;
    flag = 0;
}
```

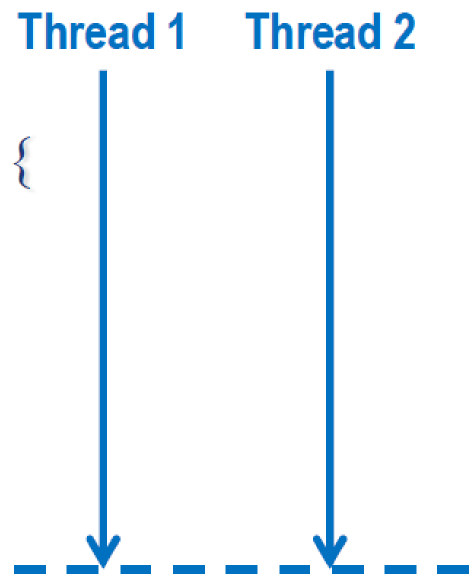
Flags Don't Guarantee Mutual Exclusion

flag

$\boxed{0}$

```
int flag = 0;

void AddHead(struct List* list, struct Node* node) {
    while (flag != 0) /* wait */;
    flag = 1;
    node->next = list->head;
    list->head = node;
    flag = 0;
}
```



Locking Mechanism

- **The previous method fails because…**
  - **(i) Checking the value of *flag* and (ii) setting its value are two distinct operations**
- **We need some sort of *atomic* test-and-set**
  - **Operating systems provide functions to do this**
- **Lock**
  - **Synchronization mechanism used to control access to shared resources**
  - **(A generic term)**

Pragma: *critical*

- **Critical section**
  - **A portion of code that only one thread at a time may execute (mutually exclusive)**
- **Syntax in OpenMP**

  #pragma omp critical
- **Good news! (^_^)**
  - **Critical sections eliminate race conditions**
- **Bad news! (@_@)**
  - **Critical sections are executed sequentially**
- **More bad news! (@_@)**
  - **You have to identify critical sections yourself**

Is the AddHead() Function Correct Now?

```
void AddHead(struct List* list, struct Node* node) {
    node->next = list->head;
#pragma omp critical
    list->head = node;
}
```

Is the AddHead() Function Correct Now?

- **You must protect both read and write access to any shared data**
- **For the AddHead() function, both lines need to be protected**

Corrected AddHead() Function

```
void AddHead(struct List* list, struct Node* node) {
#pragma omp critical
    {
        node->next = list->head;
        list->head = node;
    }
}
```

OpenMP atomic (原子) Construct

- Special case of a critical section to ensure atomic update to memory location
- Applies only to simple operations:
  - Pre- or post-increment (++)
  - Pre- or post-decrement (--)
  - Assignment with binary operator (of scalar types)
- Works on a single statement

```
#pragma omp atomic
counter += 5;
```

Critical vs. Atomic

```
#pragma omp parallel for
{
    for (i = 0; i < n; i++) {
#pragma omp critical
        x[index[i]] += WorkOne(i);
        y[i] += WorkTwo(i);
    }
}
```

- **Critical protect:**
  - **Call to WorkOne()**
  - **Finding value of index[i]**
  - **Addition of x[index[i]] and results of WorkOne()**
  - **Assignment to x array element**
- **Essentially, updates to elements in the x array are serialized**

```
#pragma omp parallel for
{
    for (i = 0; i < n; i++) {
#pragma omp atomic
        x[index[i]] += WorkOne(i);
        y[i] += WorkTwo(i);
    }
}
```

- **Atomic protects:**
  - **Addition and assignment to x array element**
- **Non-conflicting updates with be done in parallel**
- **Protection needed only if there are two threads where the index[i] values match**

```
#pragma omp parallel for
{
    for (i = 0; i < n; i++) {
#pragma omp critical
        x[index[i]] += WorkOne(i);
        y[i] += WorkTwo(i);
    }
}
```

A

```
#pragma omp parallel for
{
    for (i = 0; i < n; i++) {
#pragma omp atomic
        x[index[i]] += WorkOne(i);
        y[i] += WorkTwo(i);
    }
}
```

B

Critical vs. Atomic

```
#pragma omp parallel for
{
    for (i = 0; i < n; i++) {
#pragma omp critical
        x[index[i]] += WorkOne(i);
        y[i] += WorkTwo(i);
    }
}
```

- Critical protect:
  - Call to WorkOne()
  - Finding value of index[i]
  - Addition of x[index[i]] and results of WorkOne()
  - Assignment to x array element
- Essentially, updates to elements in the x array are serialized

```
#pragma omp parallel for
{
    for (i = 0; i < n; i++) {
#pragma omp atomic
        x[index[i]] += WorkOne(i);
        y[i] += WorkTwo(i);
    }
}
```

- Atomic protects:
  - Addition and assignment to x array element
- Non-conflicting updates with be done in parallel
- Protection needed only if there are two threads where the index[i] values match

Summary

- **Synchronization (in OpenMP)**
  - **Barrier**
    - Statement ordering among different threads
    - Any statement after the barrier will be executed after the statements before the barrier in every thread
  - **Mutual Exclusion**
    - Access ordering of shared resources
    - A mechanism to avoid race conditions
  - **Memory fence**
    - Data-related statement ordering in the same threads
    - Any data-related statement before the memory fence will be executed before the statements after the memory fence

Thank You！