

Parallel & Distributed Computing

Race Conditions and Synchronization in OpenMP

Lecture 5, Spring 2013

Instructor: 罗国杰

gluo@pku.edu.cn

Outline

◆ **Correctness issues in parallel programming (in OpenMP)**

- **Barriers**
- **Examples of race conditions**
- **Mutual Exclusion**
- **Memory fence**

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
- ...

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?

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

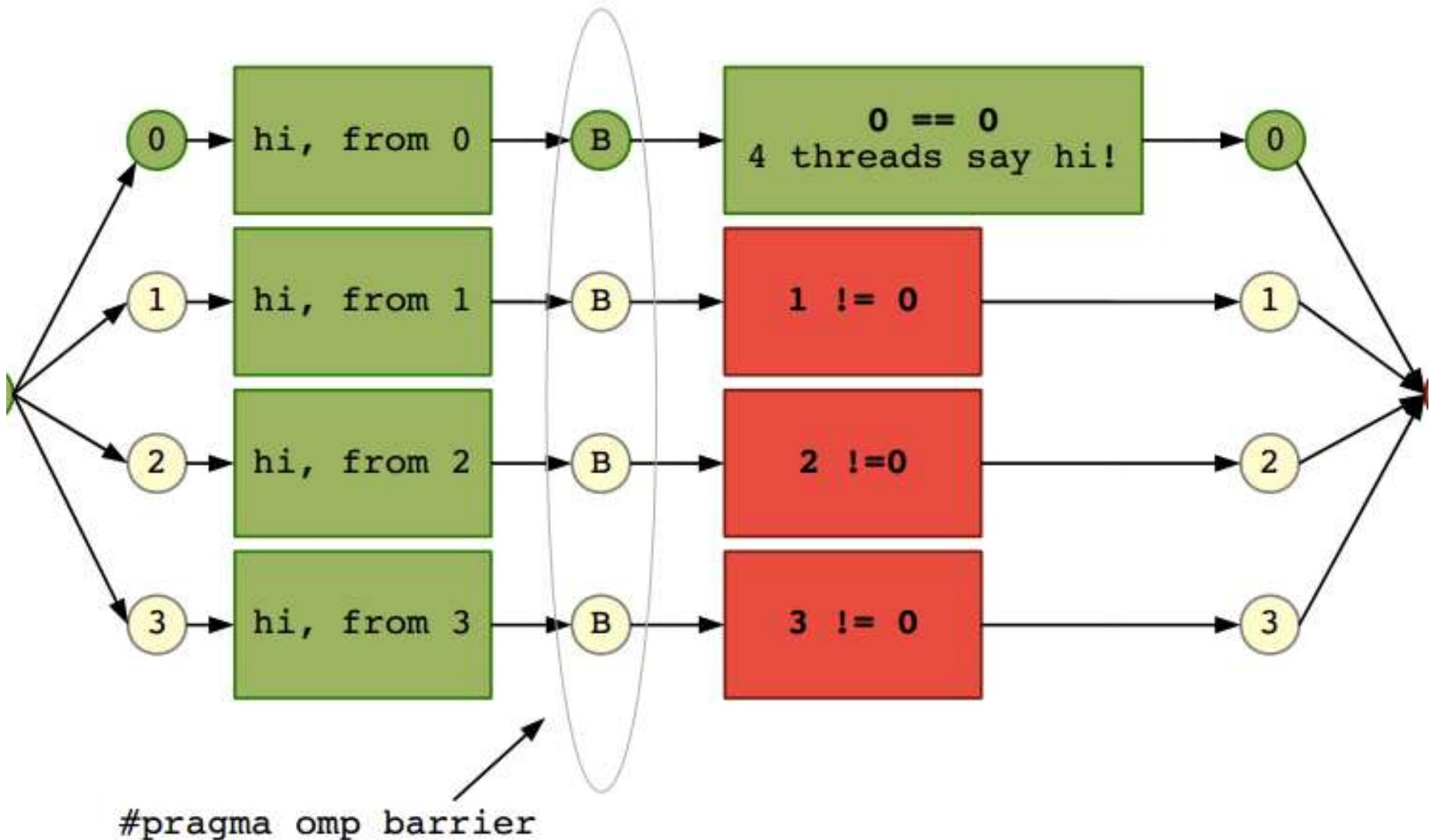
hi, from 0

hi, from 2

hi, from 1

4 threads say hi!

Example: an Explicit Barrier



Barbara Chapman, “A Guide to OpenMP,” 2010.

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);
}
```

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, pi, 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;
```

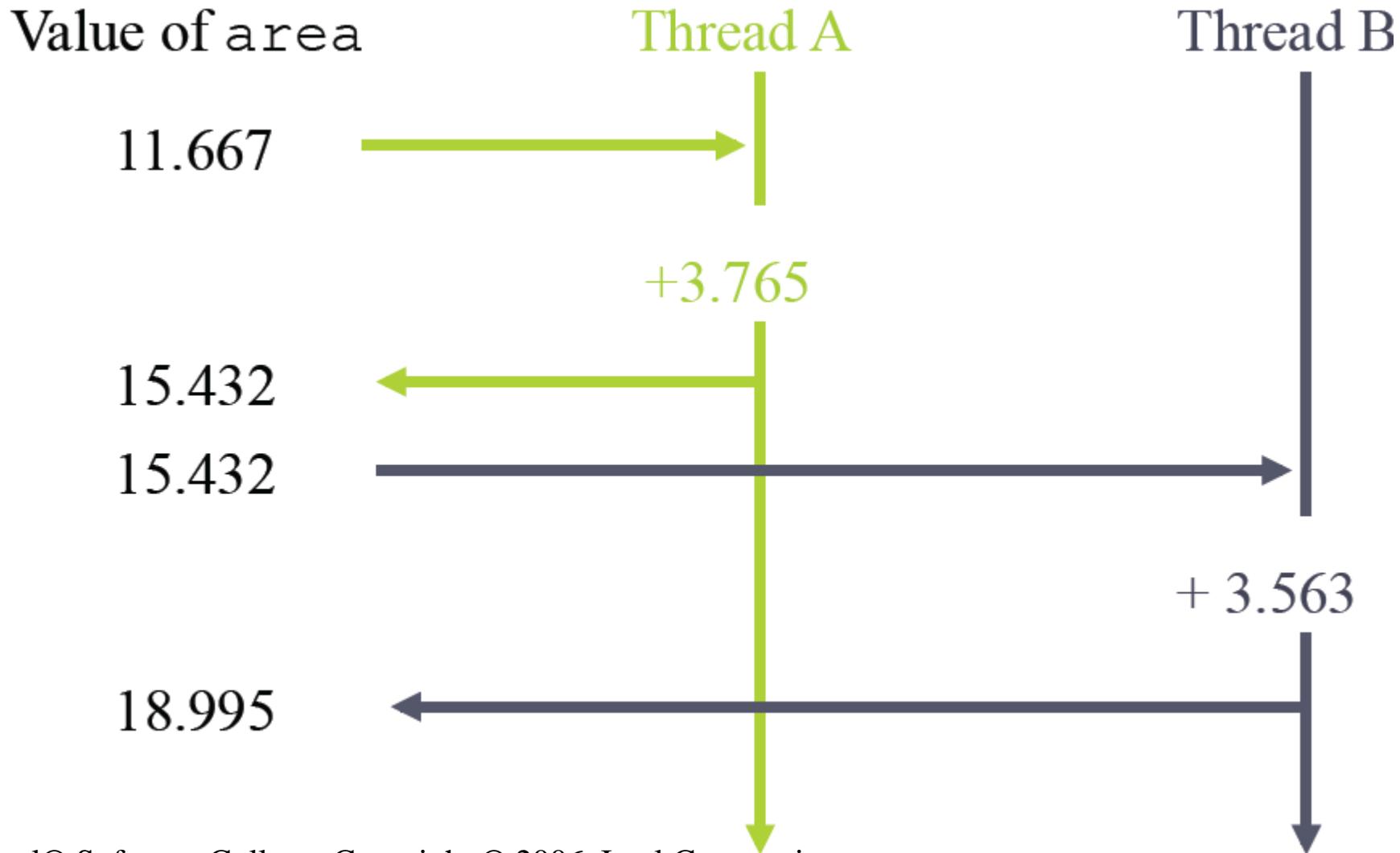
- ◆ What happens when we make the **for** loop parallel?

Race Condition

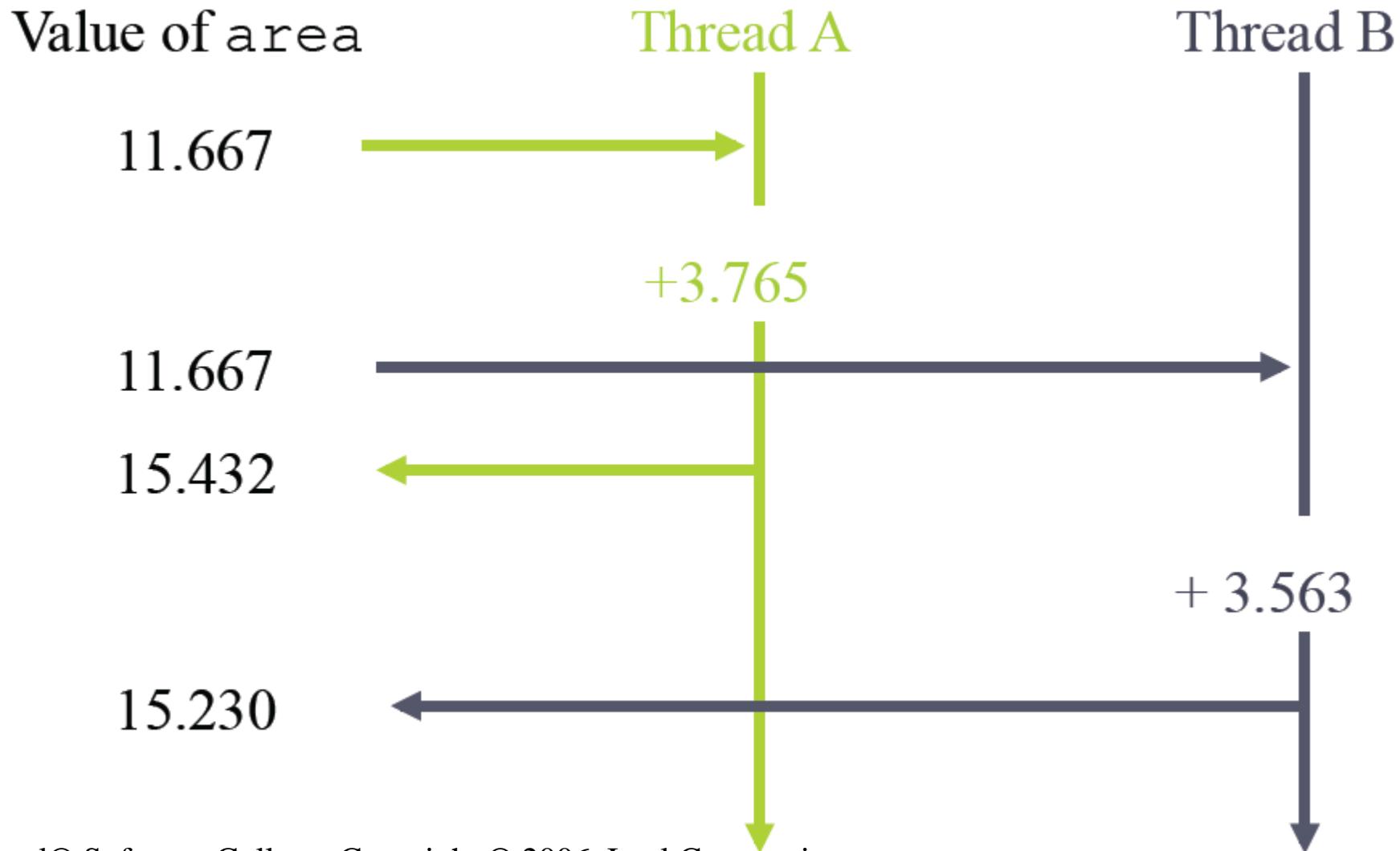
- ◆ A *race condition* is nondeterministic behavior caused by the times at which two or more threads access a shared variable
- ◆ For example, suppose both Thread A and Thread B are executing the statement...

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

One Timing \Rightarrow Correct Sum



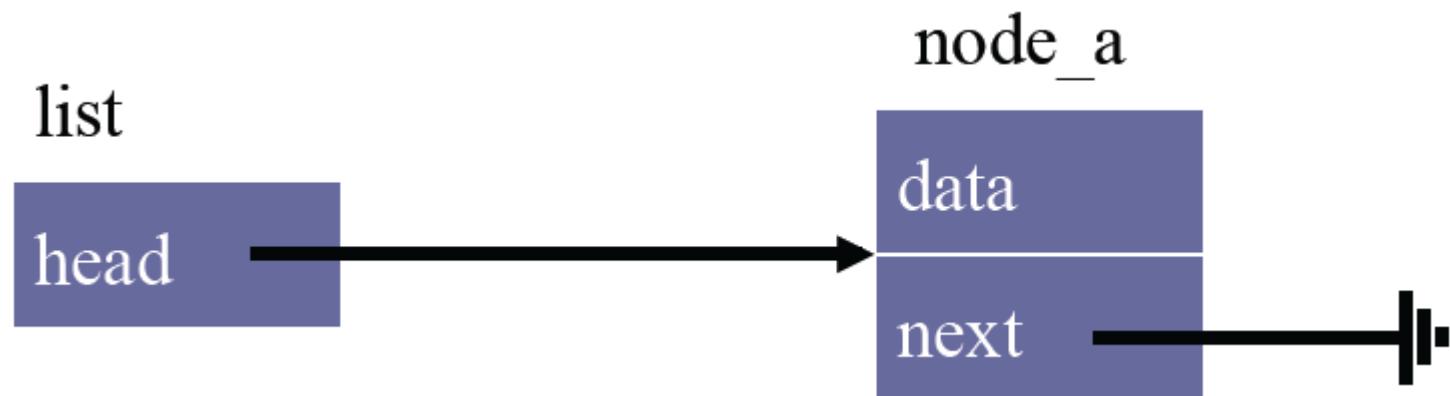
Another Timing \Rightarrow Incorrect Sum



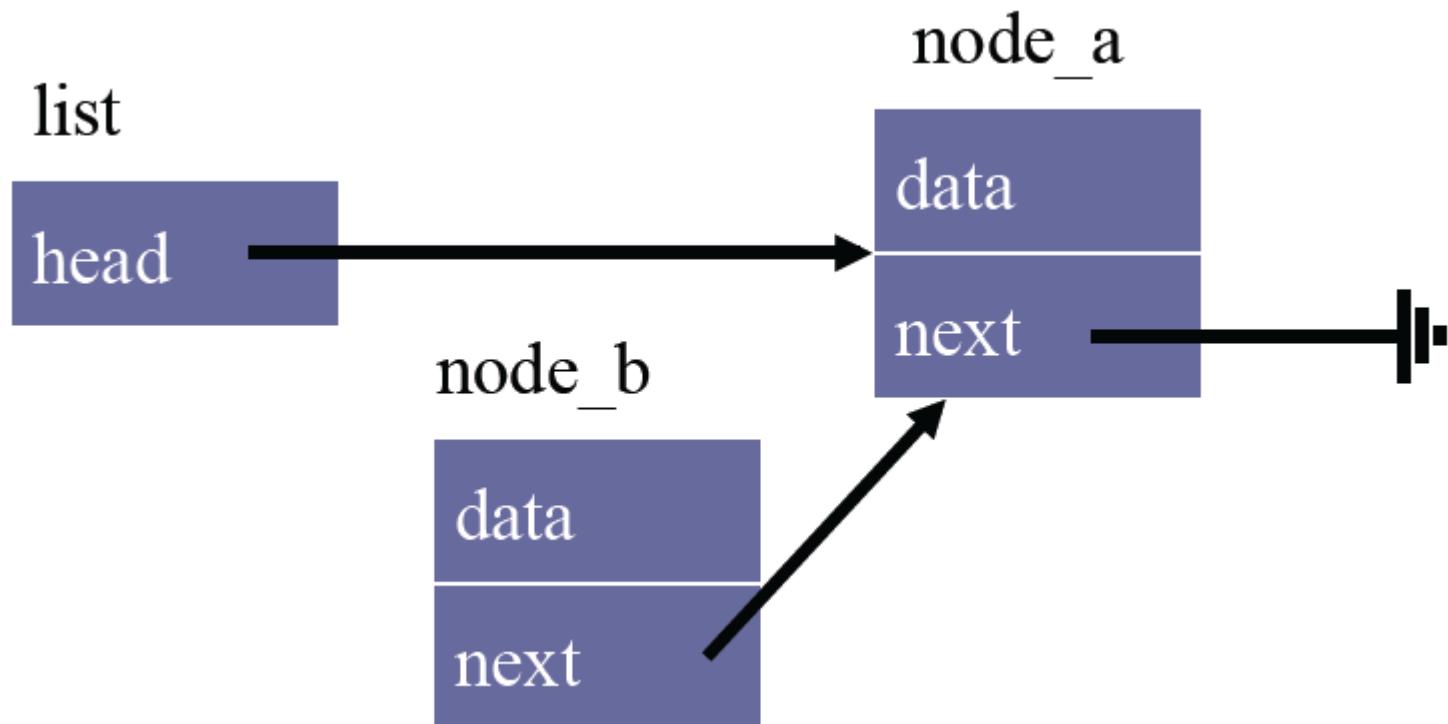
Another Race Condition Example

```
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;  
}
```

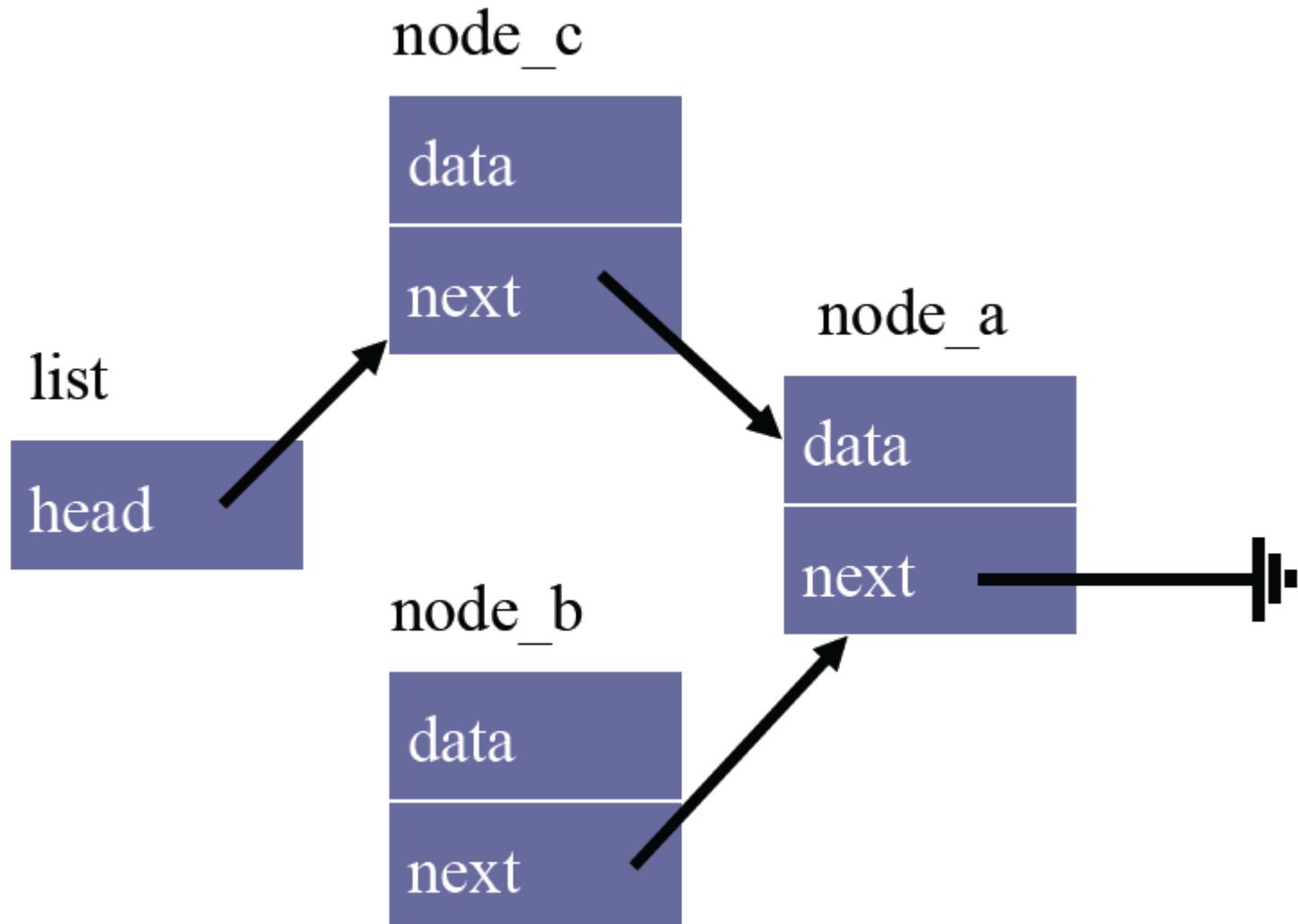
Original Singly-Linked List



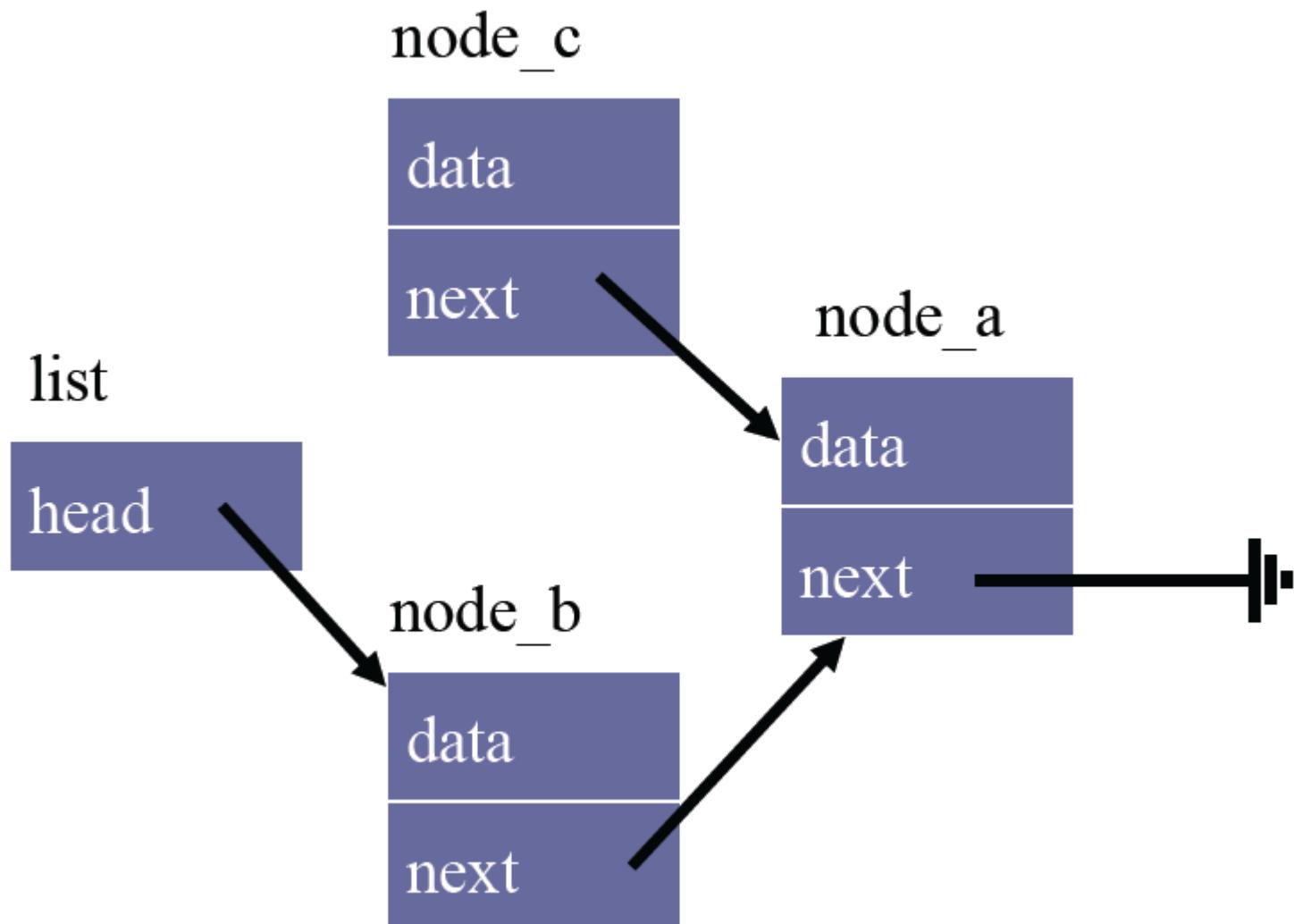
Thread 1 after Stmt. 1 of AddHead



Thread 2 Executes AddHead



Thread 1 after Stmt. 2 of AddHead



Why Race Conditions Are Nasty

- ◆ 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

```
int flag = 0;
```



Thread 1

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

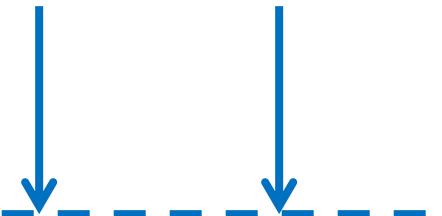
Flags Don't Guarantee Mutual Exclusion

```
int flag = 0;
```

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;  
}
```

Thread 1 Thread 2



Flags Don't Guarantee Mutual Exclusion

```
int flag = 0;
```

flag

1

```
void AddHead(struct List* list, struct Node* node) {
```

Thread 1

Thread 2



```
    while (flag != 0) /* wait */;
```

```
    flag = 1;
```

```
    node->next = list->head;
```

```
    list->head = node;
```

```
    flag = 0;
```

```
}
```

Flags Don't Guarantee Mutual Exclusion

```
int flag = 0;
```

flag
1

```
void AddHead(struct List* list, struct Node* node) {
```

```
    while (flag != 0) /* wait */;
```

```
    flag = 1;
```

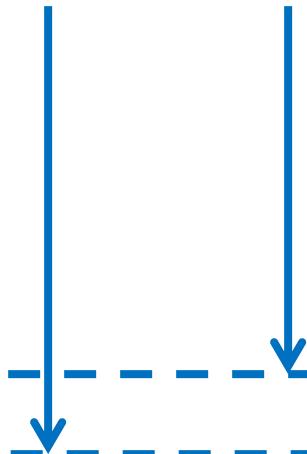
```
    node->next = list->head;
```

```
    list->head = node;
```

```
    flag = 0;
```

```
}
```

Thread 1 Thread 2



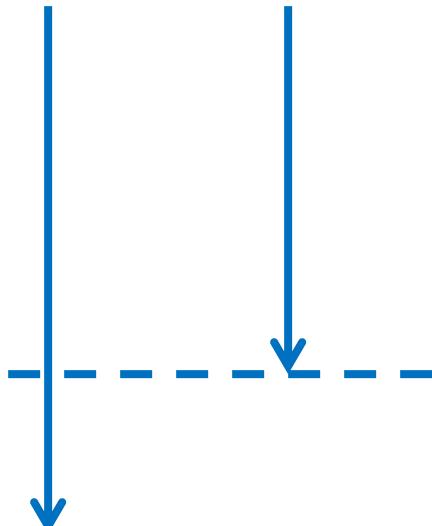
Flags Don't Guarantee Mutual Exclusion

```
int flag = 0;
```

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;  
}
```

Thread 1 Thread 2



Flags Don't Guarantee Mutual Exclusion

```
int flag = 0;
```

flag
0

```
void AddHead(struct List* list, struct Node* node) {
```

```
    while (flag != 0) /* wait */;
```

```
    flag = 1;
```

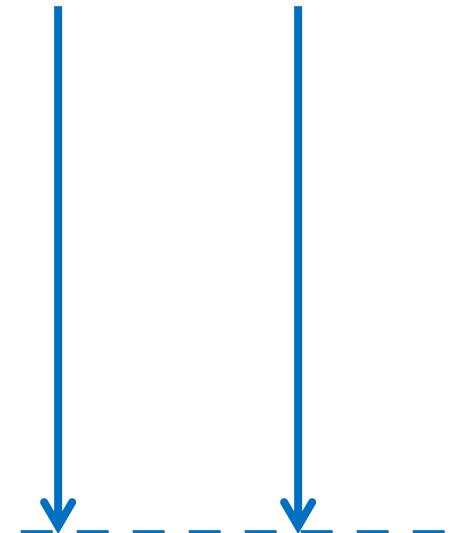
```
    node->next = list->head;
```

```
    list->head = node;
```

```
    flag = 0;
```

```
}
```

Thread 1 Thread 2



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 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;  
}
```

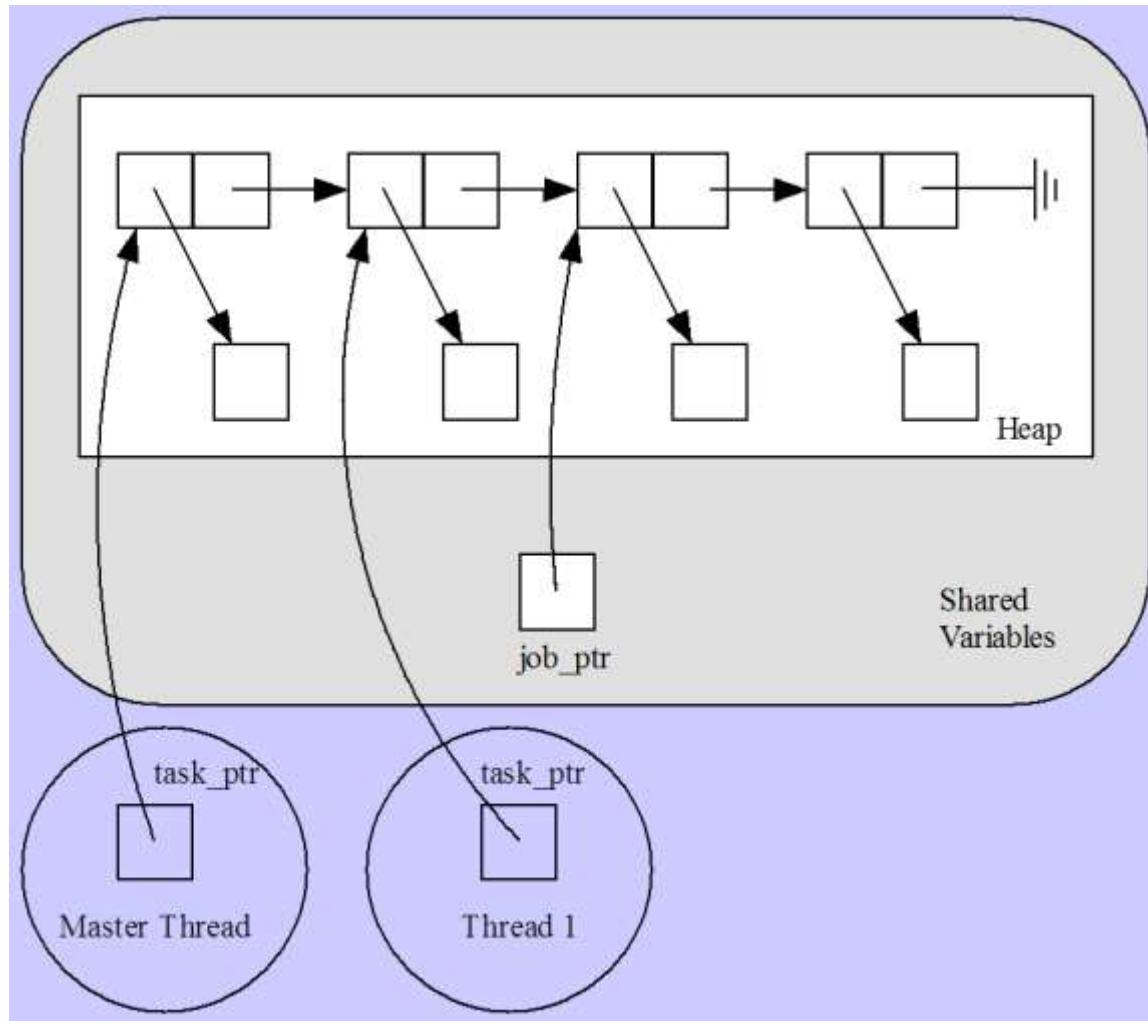
Protect All References to Shared Data

- ◆ 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;  
}  
}
```

Example: Processing a “To-Do” List



Michael J.Quinn, "Parallel Programming in C with MPI and OpenMP," 2003.

“To-Do” List: Sequential Code (1/2)

```
int main(int argc, char* argv[]) {  
    struct task_struct* job_ptr;  
    struct task_struct* task_ptr;  
    ...  
    task_ptr = get_next_task(&job_ptr);  
    while (task_ptr != NULL) {  
        complete_task(task_ptr);  
        task_ptr = get_next_task(&job_ptr);  
    }  
}
```

“To-Do” List: Sequential Code (2/2)

```
struct task_struct* get_next_task(struct task_struct** job_ptr) {  
    struct task_struct* answer;  
    if (*job_ptr == NULL) answer = NULL;  
    else {  
        answer = (*job_ptr)->task;  
        *job_ptr = (*job_ptr)->next;  
    }  
    return answer;  
}
```

Parallelization Strategy

- ◆ Every thread should repeatedly take next task from list and complete it, until there are no more tasks
- ◆ We must ensure no two threads take the same task from the list; i.e., must declare a critical section

“To-Do” List: Use of parallel Pragma

```
#pragma omp parallel private(task_ptr)
{
    task_ptr = get_next_task(&job_ptr);
    while (task_ptr != NULL) {
        complete_task(task_ptr);
        task_ptr = get_next_task(&job_ptr);
    }
}
```

◆ Almost correct...

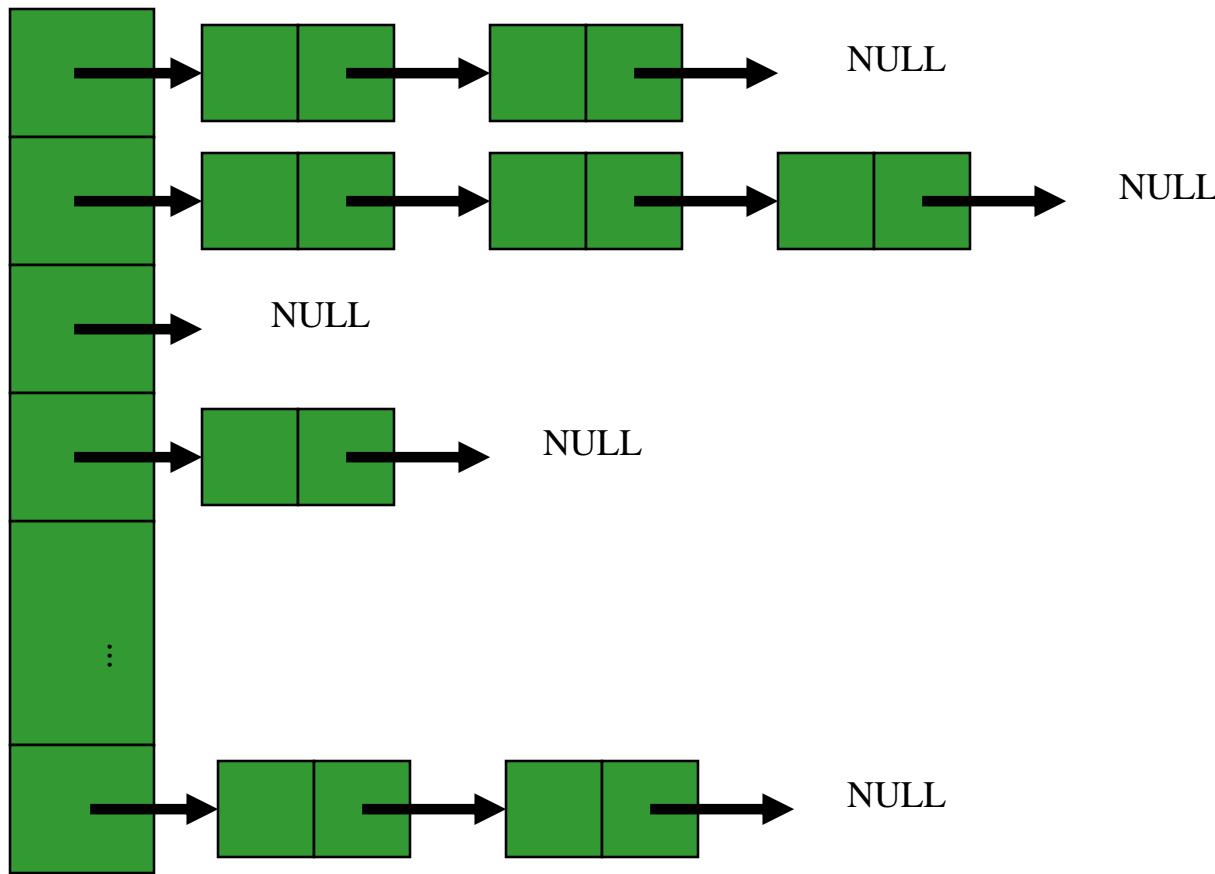
Critical Section for get_next_task

```
struct task_struct* get_next_task(struct task_struct** job_ptr) {  
    struct task_struct* answer;  
#pragma omp critical  
    {  
        if (*job_ptr == NULL) answer = NULL;  
        else {  
            answer = (*job_ptr)->task;  
            *job_ptr = (*job_ptr)->next;  
        }  
    }  
    return answer;  
}
```

Important: Lock Data, Not Code

- ◆ Locks should be associated with data objects
- ◆ Different data objects should have different locks

Example: Hash Table Creation



Locking Code: Inefficient

```
#pragma omp parallel for private (index)
for (i = 0; i < elements; i++) {
    index = hash(element[i]);
    #pragma omp critical
    insert_element (element[i], index);
}
```

Locking Data: Efficient

```
/* Static variable */  
omp_lock_t hash_lock[HASH_TABLE_SIZE];
```

Declaration

```
/* Inside function 'main' */  
for (i = 0; i < HASH_TABLE_SIZE; i++)  
    omp_init_lock(&hash_lock[i]);
```

Initialization

```
void insert_element (ELEMENT e, int i)  
{  
    omp_set_lock (&hash_lock[i]);  
    /* Code to insert element e */  
    omp_unset_lock (&hash_lock[i]);  
}
```

Use

OpenMP atomic 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 will be done in parallel
- Protection needed only if there are two threads where the index[i] values match

Reminder: Motivating Example

```
double area, pi, 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;
```

◆ Where is the critical section?

Solution #1

```
double area, pi, tmp, x;  
int i, n;  
...  
area = 0.0;  
#pragma omp parallel for private(x)  
for (i = 0; i < n; i++) {  
    x = (i + 0.5) / n;  
#pragma omp critical  
    area += 4.0 / (1.0 + x*x);  
}  
pi = area / n;
```

- ◆ **This ensures area will end up with the correct value**
- ◆ **How can we do better?**

Solution #2

```
double area, pi, tmp, x;  
int i, n;  
...  
area = 0.0;  
#pragma omp parallel for private(x, tmp)  
for (i = 0; i < n; i++) {  
    x = (i + 0.5) / n;  
    tmp = 4.0 / (1.0 + x*x);  
#pragma omp critical  
    area += tmp;  
}  
pi = area / n;
```

- ◆ This reduces amount of time spent in critical section
- ◆ How can we do better?

Solution #3

```
double area, pi, tmp, x;  
int i, n;  
...  
area = 0.0;  
#pragma omp parallel private(tmp)  
{  
    tmp = 0.0;  
#pragma omp for private(x)  
    for (i = 0; i < n; i++) {  
        x = (i + 0.5) / n;  
        tmp += 4.0 / (1.0 + x*x);  
    }  
#pragma omp critical  
    area += tmp;  
}  
pi = area / n;
```

- ◆ Why is this better?
- ◆ Can we still do better?

Solution #4 Use Reduction

```
double area, pi, x;  
int i, n;  
...  
area = 0.0;  
#pragma omp parallel for private(x) \  
           reduction(+:area)  
for (i = 0; i < n; i++) {  
    x = (i + 0.5)/n;  
    area += 4.0/(1.0 + x*x);  
}  
pi = area / n;
```

Memory Fences in OpenMP

◆ Syntax

```
#pragma omp flush (<variable-list>)
```

- All the shared variables will be flushed without a list
- Automatically inserted at most points where it is needed
 - at entry to and exit from *parallel*
 - at exit from *for*
 - at entry to and exit from *parallel for*
 - *barrier*
 - ...
- ◆ identifies a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory

Example: flush Pragma

```
int main() {
    int data;
    int flag = 0;
    #pragma omp parallel sections \
    num_threads(2)
    {
        #pragma omp section
        {
            printf_s("Thread %d: ",
            omp_get_thread_num());
            read(&data);
            #pragma omp flush(data)
            flag = 1;
            #pragma omp flush(flag)
            // Do more work.
        }
    }
}
```

```
#pragma omp section
{
    while (!flag) {
        #pragma omp flush(flag)
    }
    #pragma omp flush(data)

    printf_s("Thread %d: ",
    omp_get_thread_num());
    process(&data);
    printf_s("data = %d\n", data);
}
}
```

Implicit flush at ...

- ◆ **barrier**
- ◆ **parallel - upon entry and exit**
- ◆ **critical - upon entry and exit**
- ◆ **ordered - upon entry and exit**
- ◆ **for - upon exit**
- ◆ **sections - upon exit**
- ◆ **single - upon exit**

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