

Multithreading Applications in Win32

(Chapter4. Synchronization)

Myung-Hun , Kang
freehuni@mmlab.net

[Multimedia Lab.](#)

Dept. of Electrical and Computer Eng.
University of Seoul
Seoul, Korea

Contents

4.0 Main Objectives

4.1 Overview

4.2 Critical Sections

4.3 Deadlock

4.4 The Dining Philosophers

4.5 Mutexes

4.6 Semaphores

4.7 Event Objects

4.8 Displaying Output from Worker Threads

4.9 Interlocked Variables

4.10 Summary of Synchronization Mechanisms

4.0 Main Objectives

- The Win32 **synchronization mechanisms** in multitasking environment.

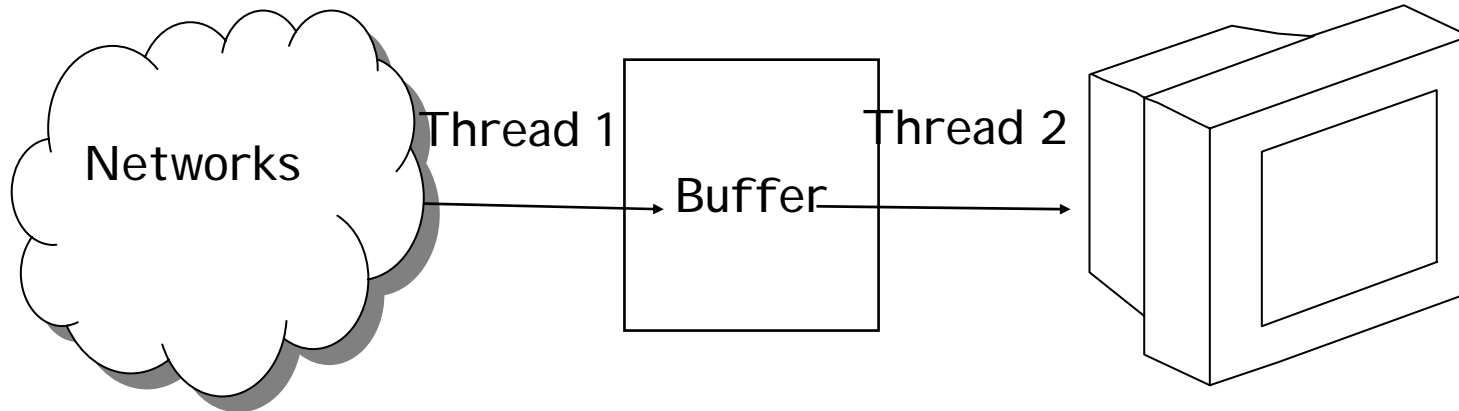
4.1 Overview

- The **coordination of threads and processes** in Win32 is performed by **synchronization mechanisms**. (= traffic light for threads.)
- **Critical Section** (single process; non kernel object)

:

Thread가

Resource



4.1 Overview

- **Mutexes**

: Mutually Exclusive

Critical Section + Multi Processes

mutex critical section

Threads

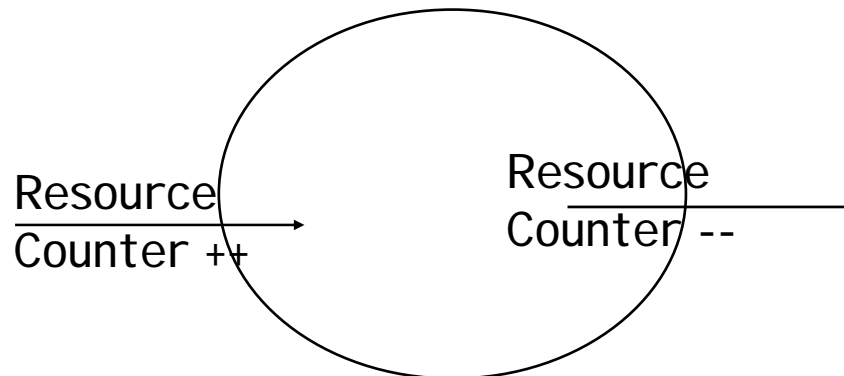
: "Abandoned ?"

- **Semaphores**

:Resource

Threads

Semaphores



4.1 Overview

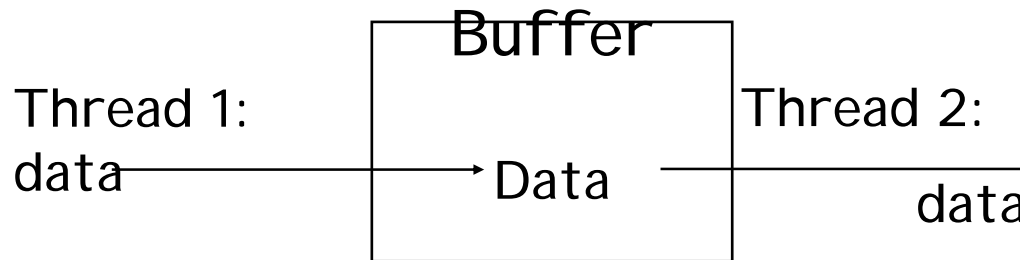
- Event Objects

:

. (Semaphore
Signal

Event
.)

Threads
Mutex



- Resource

: memory location, data structure, file...

4.2 Critical Sections

- Portion of a code that accesses a shared resource.
- ~is used to enforce **mutual exclusion** between threads **within a single process**.
- Only one Thread at a time is allowed to be “inside” the critical section.(permission)
- Local object (not kernel object -> handle
InitializeCriticalSection())

4.2 Critical Sections

- VOID [InitializeCriticalSection](#)(
LPCRITICAL_SECTION lpCriticalSection
);

- VOID [DeleteCriticalSection](#)(
LPCRITICAL_SECTION lpCriticalSection
);

4.2 Critical Sections

- VOID [EnterCriticalSection](#)(
LPCRITICAL_SECTION lpCriticalSection
);
- VOID [LeaveCriticalSection](#)(
LPCRITICAL_SECTION lpCriticalSection
);

4.2 Critical Sections

- Example : Linked list (P.79)

```
typedef struct _Node{
    struct _Node *next;
    int data;
} Node;
typedef struct _List{
    Node *head;
    CRITICAL_SECTION Critical_sec;
} List;
List *CreateList(){
    List *pList = malloc(sizeof(pList));
    pList->head = NULL;
    InitializeCriticalSection(&pList->Critical_sec);
    return pList;
}
```

4.2 Critical Sections

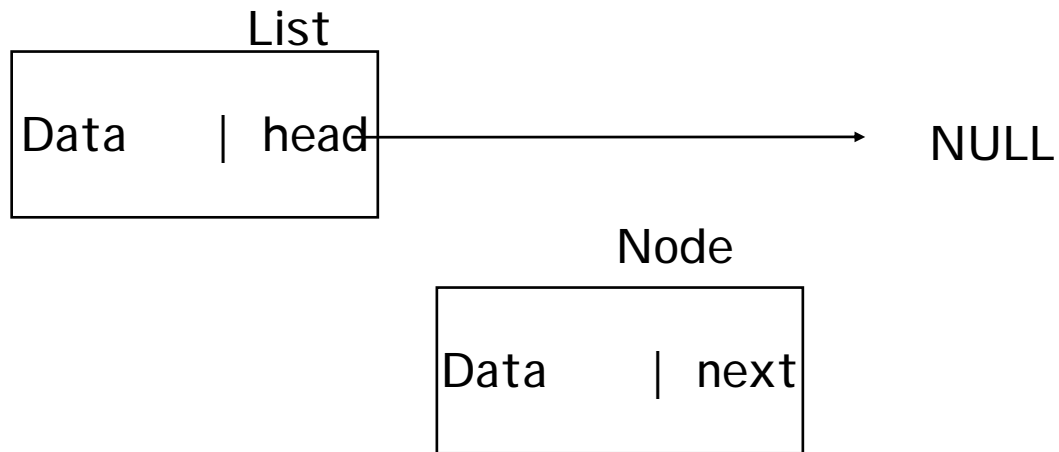
```
void DeleteList(List *pList)
{
    DeleteCriticalSection(&pList->Critical_sec);
    free(pList);
}
```

```
void AddHead(List *pList, Node *node)
{
    EnterCriticalSection(&pList->critical_sec);
    node->next = pList->head;
    pList->head = node;
    LeaveCriticalSection(&pList->Critical_sec);
}
```

4.2 Critical Sections

- Example : Linked list (P.79)

AddHead()



4.2 Critical Sections

```
Void Insert(List *pList, Node *afterNode, Node *newnode){
    EnterCriticalSection(&pList->critical_sec);
    if(afterNode == NULL){
        AddHead(pList, newNode);
    }
    else{
        newNode->next = afterNode->next;
        afterNode->next = newNode;
    }
    LeaveCriticalSection(&pList->Critical_sec);
}

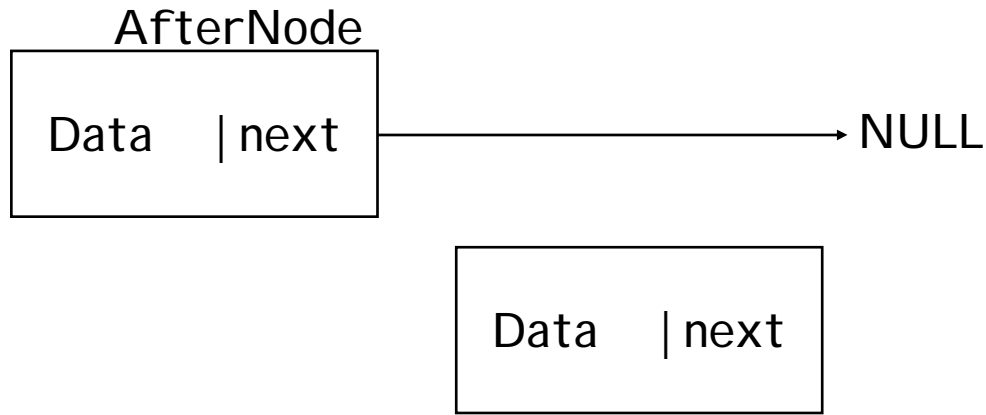
Node *next(List *pList, Node *node){
    Node* next;
    EnterCriticalSection(&pList->Critical_sec);
    next = node->next;
    LeaveCriticalSection(&pList->Critical_sec);
    return next;
}
```

4.2 Critical Sections

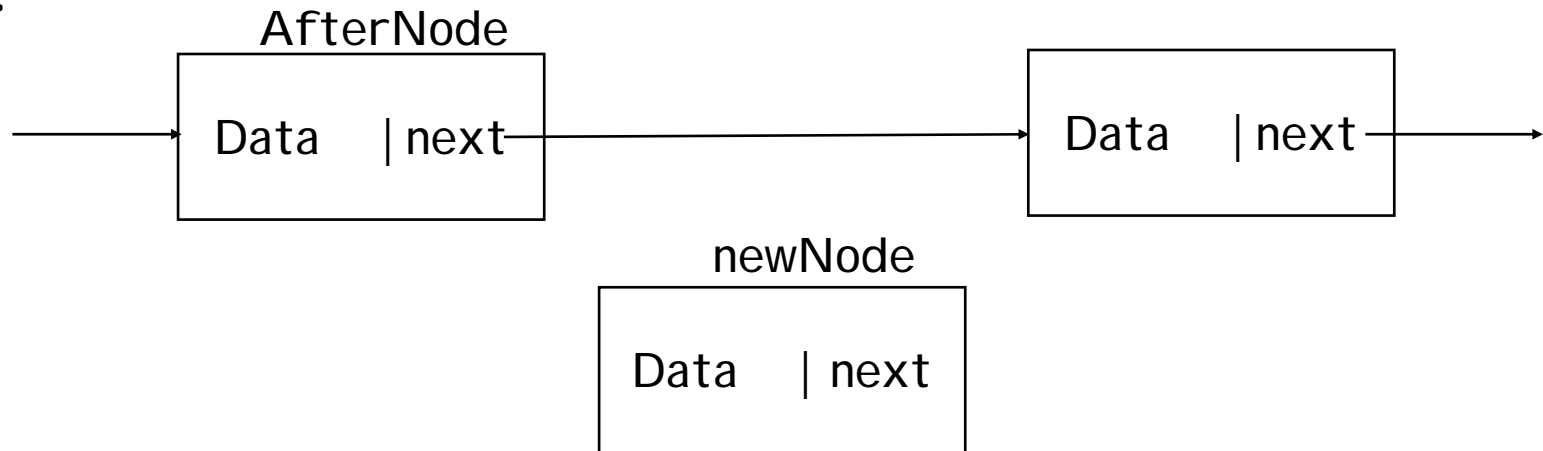
- Example : Linked list (P.79)

Insert()

1.



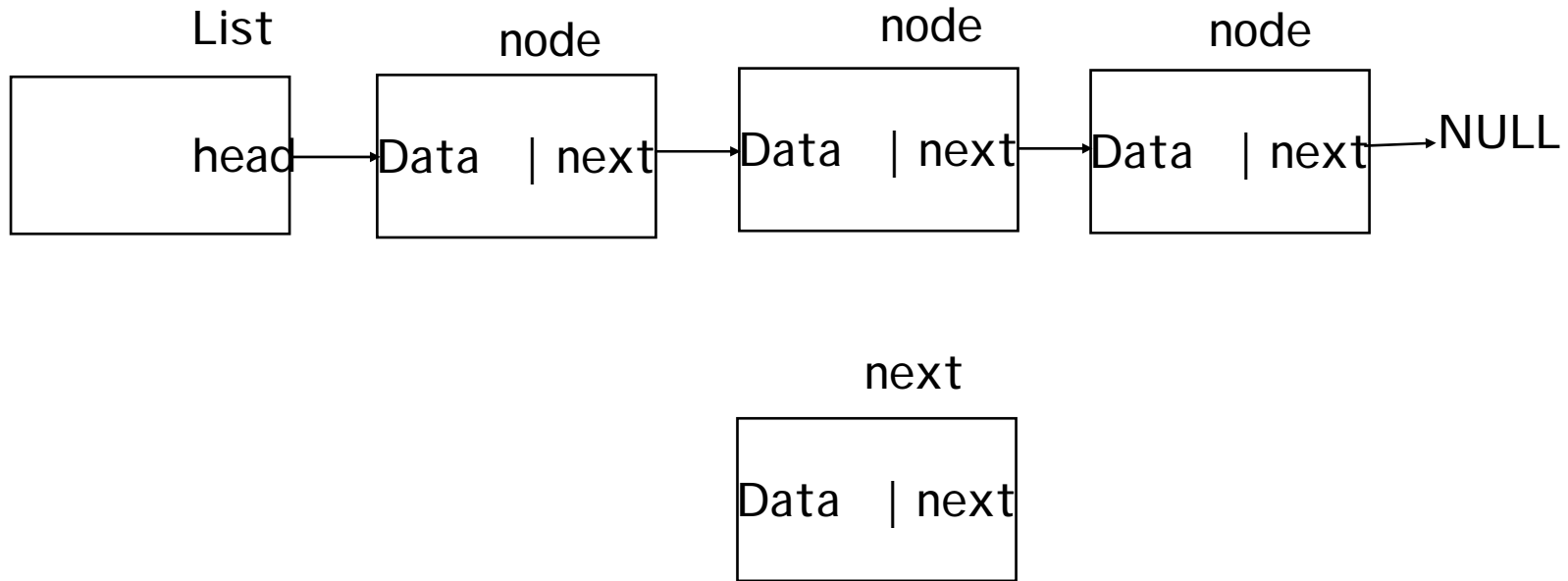
2.



4.2 Critical Sections

- Example : Linked list (P.79)

*Next()



4.2 Critical Sections

- Minimizing Lock Time
 - Don't lock a resource for long time !
 - Never call Sleep() or any of the Wait...() APIs inside of a critical section.
 - How often the resource will be used and how quickly a thread must release the resource.
- Void Dangling Critical Sections
 - LeaveCriticalSection() thread가 ..?
 - Critical section is not kernel object -> there is no way to tell if the thread currently inside a critical section is still alive.

4.3 Deadlock

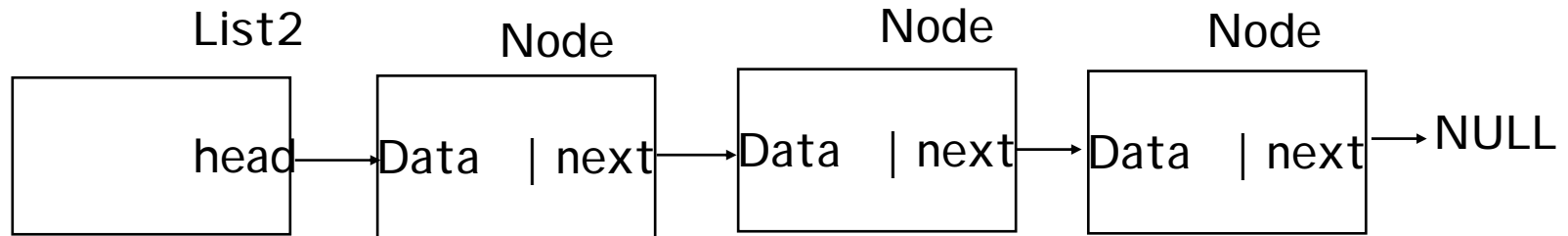
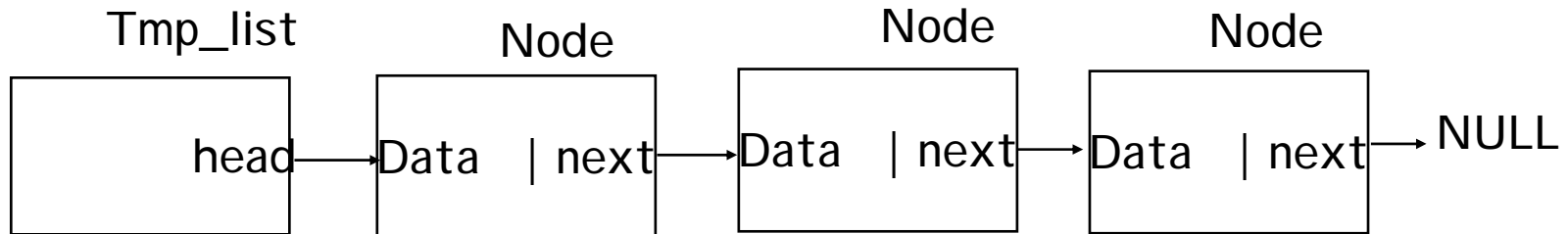
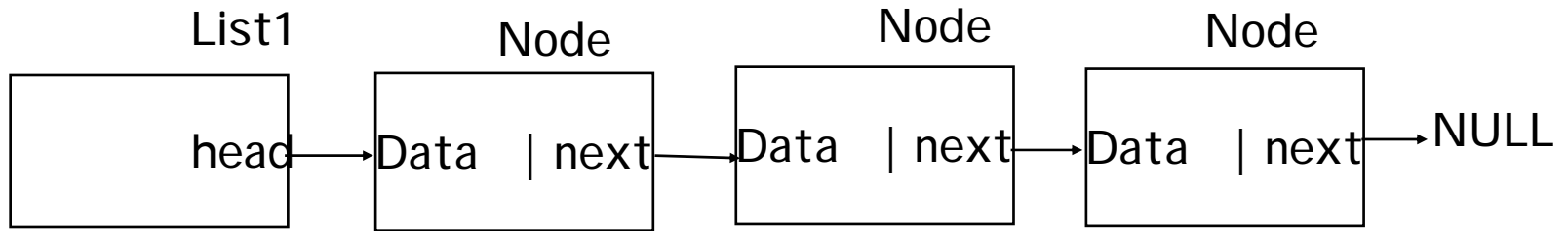
Example SwapLists() -p.82

```
void SwapLists(List *list1, List *list2){
    List *tmp_list;
    EnterCriticalSection(list1->critical_sec); //context switch
    EnterCriticalSection(list2->critical_sec);
    tmp_list = list1->head;
    list1->head = list2->head;
    list2->head = tmp_list;
    LeaveCriticalSection(list1->critical_sec);
    LeaveCriticalSection(list2->critical_sec);
}
```

ThreadA SwapLists(home_address_list, work_address_list);

ThreadA SwapLists(work_address_list, home_address_list);

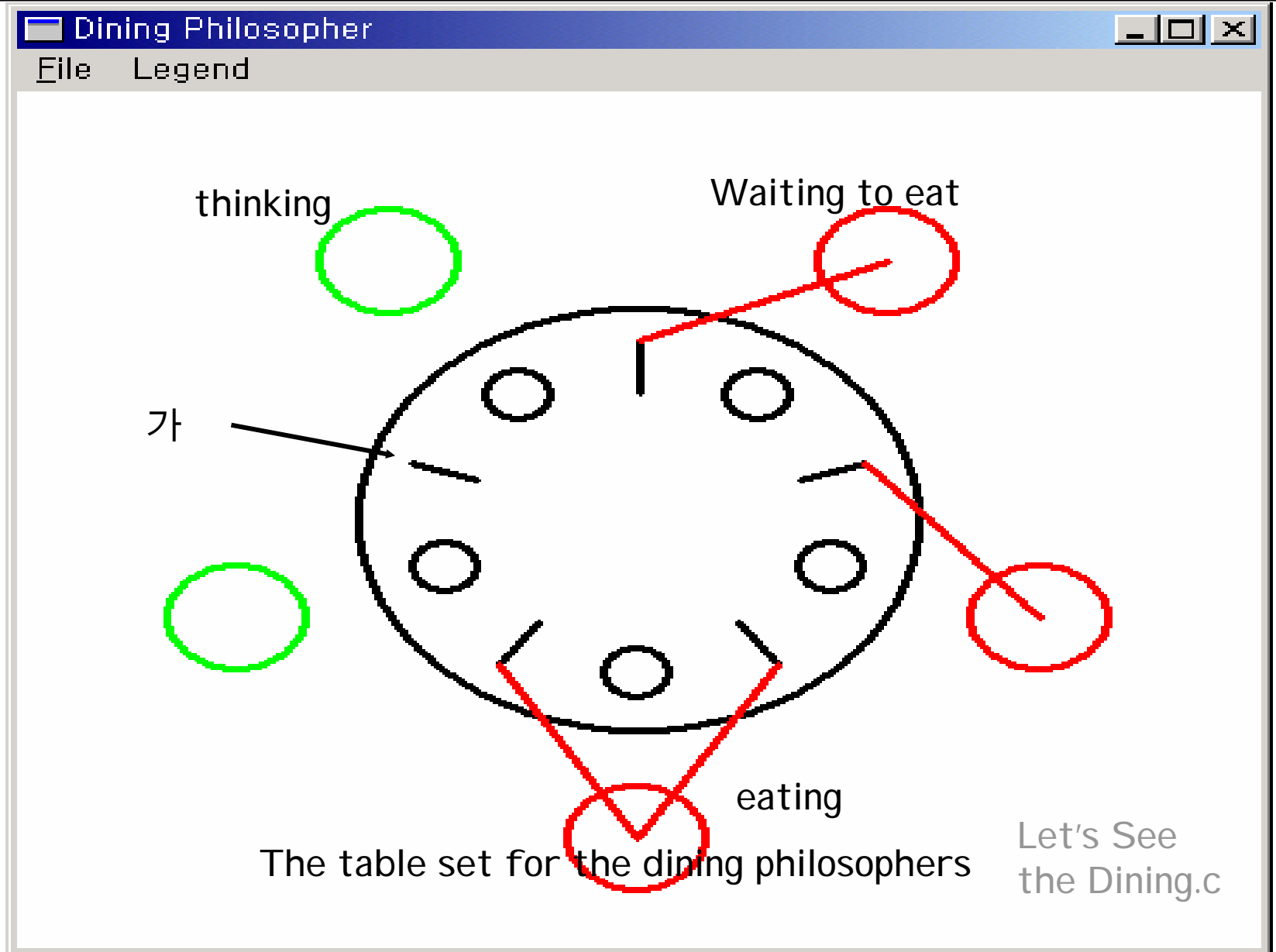
4.3 Deadlock



4.3 Deadlock

- Deadlock() = “The Deadly Embrace”
- Deadlock will not happen by allocating all the resources you need as a single operation with `WaitForMultipleObjects()`. (later...-> P.91)
- All-or-nothing proposition

4.4 The Dining Philosophers



4.5 Mutexes

- MUTual Exclusion
- Only one thread at a time is allowed to own a mutex, just as only one thread at a time can enter a critical section.
- A kernel object that will enforce mutual exclusion between threads even if they are in different processes.

Mutexes	Critical section
Almost 100 times longer	
Between Processes	Within the same process
Timeout (see Wait...() func)	not

4.5 Mutexes

- The comparison of the function

CRITICAL_SECTION	Mutex Kernel Object
InitializeCriticalSection()	CreatMutex() OpenMutex()
EnterCriticalSection()	WaitForSingleObject() WaitForMultipleObjects() MsgWaitForMultipleObjects()
LeaveCriticalSection()	ReleaseMutex()
DeleteCriticalSection()	CloseHandle()

4.5 Mutexes

- **Creating Mutex**

```
HANDLE CreateMutex(  
    LPSECURITY_ATTRIBUTES lpAttribute,  
    BOOL bInitialOwner,  
    LPCTSTR lpName  
);
```

- lpAttribute: Use NULL to get the default Security Attributes.
 - bInitialOwner: Set to TRUE if you want the thread that called CreateMutex() to own the mutex.
 - lpName: Any thread or process can refer to this mutex by name
- a Mutex has a reference count. You should call CloseHandle()
 - CreateMutex Reference Count++
 - CloseHandle Reference Count--

4.5 Mutexes

- **Opening Mutex**

```
HANDLE OpenMutex(  
    DWORD dwDesiredAccess, // access  
    BOOL bInheritHandle, // inheritance option  
    LPCTSTR lpName // object name );
```

- dwDesiredAccess : Specifies the requested access to the mutex.
- bInheritHandle : Specifies whether the returned handle is inheritable.
- lpName: Name to access Mutex.
- Mutex has a reference count. You should call CloseHandle()

-

Mutex Open

.

4.5 Mutexes

- Locking a Mutex
 - To acquire ownership of the mutex, use one of the Win32 Wait...() functions.
 - If a thread **waits on a non-signaled mutex** then the thread will **block**.
 - A Mutex is signaled when no thread owns it.
 - Scenario (textbook p.87 side effect)

BOOL [ReleaseMutex](#)(HANDLE hMutex);

- a thread releases ownership of the mutex by calling `ReleaseMutex()`.

4.5 Mutexes

- Handling Abandoned Mutexes
 - If a thread that owns a mutex exits without calling `ReleaseMutex()`, the mutex is not destroyed.
 - Mutex -> unowned, nonsignaled
 - `Wait...()` -> return : `WAIT_ABANDONED_0`
(`~WAIT_ABANDONED_n - 1`)
- Letting the Philosophers Eat
 - Source

4.5 Mutexes

- Fixing SwapLists

(Ex. Listing 4-2 p.90)

```
Void SwapLists(struct List *list1, struct List *list2)
```

```
{  
    struct List *tmp_list;  
    HANDLE arrhandles[2];  
    arrHandles[0] = list1->hMutex;  
    arrHandles[1] = list2->hMutex;  
    WaitForMultipleObjects(2, arrHandles, TRUE, INFINITE);  
    tmp_list = list1->head;  
    list1->head = list2->head;  
    list2->head = tmp_list;  
    ReleaseMutex(arrHandles[0]);  
    ReleaseMutex(arrHandles[1]);  
}
```

4.5 Mutexes

- Why Have an Initial Owner?
 - bInitialOwner of CreateMutex()
 - It prevents a race condition.

```
HANDLE hMutex = CreateMutex(NULL, FALSE, "Simple Name"); <<  
int result = WaitForSingleObject(hMutex, INFINITE);
```

4.6 Semaphores

- They are the key ingredient in solving various producer/consumer problems where a buffer is being read and written at the same time.
- A semaphore in Win32 may be locked at most n times, where n is specified when the semaphore is created.
- EX) car rental problem.

(convertible = resource, = resource thread)

1. 3 convertibles, 4 agents, convertible 4

2. agents 3 .(..)

3. !

3. key , 4 1 Key !

4.6 Semaphores

- **Solution 1.** 1 1 agent (= mutex)

->

- **Solution 2.** 1 agent

->

- **Solution 3.** Semaphore

convertibles essentially identical.

가 convertible semaphore

agent count

semaphore count가 0

- Maximum Count 1 Mutex
->(Mutex is often referred to as a **binary semaphore**)

4.6 Semaphores

- **Creating Semaphores**

HANDLE CreateSemaphore(

LPSECURITY_ATTRIBUTES IpAttribute,

LONG InitialCount,

LONG MaximumCount,

LPCTSTR IpName

);

- IpAttribute: Use NULL to get the default Security Attributes.
- InitialCount: Specifies an initial count for the semaphore object (0 <= this <= MaximemCount)
- MaximumCount: Specifies the maximum count for the semaphore object
- IpName: Any thread or process can refer to this Semaphore by name

4.6 Semaphores

- Acquiring Locks
 - The Value of the semaphore represents the number of resources currently available.
 - You acquire a lock on a semaphore by using any of the Wait...()
 - There is no concept of ownership (Semaphore has no owner)
 - Several Threads can lock a semaphore at the same time
 - Semaphore can be released by any thread

4.6 Semaphores

- Releasing Locks
 - To release a lock, call `ReleaseSemaphore()`
 - This function increments the semaphore's value

```
BOOL ReleaseSemaphore(  
    HANDLE hSemaphore,  
    LONG IReleaseCount,  
    LPLONG lpPreviousCount  
);
```

`hSemaphore` Handle to the semaphore

`IReleaseCount` Increment the value of the semaphore

`lpPreviousCount` Return the previous value of the semaphore.

Note the there is no way to get the current value.

4.6 Semaphores

- Why Have an Initial Count?
 - Initial Value of the semaphore
 - Must be greater than or equal to zero and less than MaximumCount
 - ReleaseSemaphore() can increment the count up to its maximum

4.7 Event Objects

- The most flexible type of synchronization object in Win32
- Kernel Object (signaled or nonsignaled)

```
HANDLE CreateEvent(  
    LPSECURITY_ATTRIBUTES lpEventAttributes,  
    BOOL bManualReset,           // AutoReset or Manual Reset  
    BOOL bInitialState,         // initial state  
    LPCTSTR lpName              // object name  
);
```

SetEvent() Set the event object to the signaled state

ResetEvent() Set the event object to the nonsignaled state

PulseEvent() **Manual Reset Event:** Set the event object to the signaled state, wake everything up that is currently waiting, *then return to the nonsignaled state.*

Auto Reset Event: Set the event object to the signaled state, wake up a single waiting thread (if any), *return to the nonsignaled state.*

4.7 Event Objects

- Use Wait...() functions to block
- In Automatic the event object is always nonsignaled!
-> ResetEvent does nothing!

4.8 Displaying Output from Worker Threads

- SendMessage(...)
 - 가 .
- PostMessage(...)
 - MessageQueue .

4.9 Interlocked Variables

- The simplest type of synchronization mechanism.
- Operate on a standard 32-bit long variable.
- No ability to wait

- `InterlockedIncrement(LPLONG lpTarget)`
- `InterlockedDecrement(LPLONG lpTarget)`
 - These functions return a comparison against zero.
 - Use these functions to maintain your object reference count.
(ex: `AddRef()`, `Release()`)

4.10 Summary of Synchronization Mechanisms

- **Critical Section**
 - Is a local object, not a kernel object
 - Is fast and efficient
 - Cannot be waited on more than one at a time
 - Cannot determine if it was abandoned by a thread
- **Mutex**
 - Is a kernel object
 - Generates an 'abandoned' error if the owning thread terminates
 - Can be used in Wait...() calls
 - Is Named, and can be opened across processes.
 - Can only be released by the thread that owns it

4.10 Summary of Synchronization Mechanisms

- Semaphore
 - Is a kernel object
 - Has no owner
 - Is Nameed
 - Can be released by any thread
- Event Object
 - Is a kernel object
 - Is completely under program control
 - Is good for designing new synchronization objects
 - Does not queue up wake-up requests
 - Is Named
- Interlocked Variable
 - Useful for reference counting