

Multithreading Applications in Win32 (Chapter4. Synchronization)

Seong Jong Choi
chois@mmlab.net

[Multimedia Lab.](#)

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

Contents

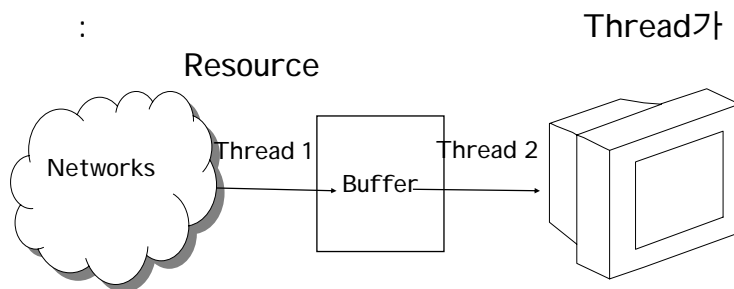
- 4.0 Main Objectives
- 4.1 Overview
- 4.2 **Critical Sections**
- 4.3 Deadlock
- 4.4 The Dining Philosophers
- 4.5 **Mutexes**
- 4.6 **Semaphores**
- 4.7 **Event Objects**
- 4.8 Displaying Output from Worker Threads
- 4.9 Interlocked Variables
- 4.10 Summary of Synchronization Mechanisms

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)



4.1 Overview

- **Mutexes**

: Mutually Exclusive
Critical Section + Multi Processes
mutex critical section

: "Abandoned ?"

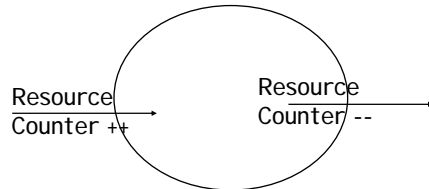
Threads

- **Semaphores**

:Resource

Semaphores

Threads



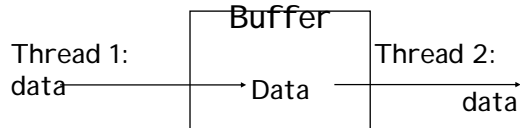
4.1 Overview

- **Event Objects**

: Semaphore
Signal

Event
(.)

Threads
Mutex



- **Resource**

: memory location, data structure, file...

4.2 Critical Sections

- Critical Section
 - Portion of a code that accesses a shared resource.
- Critical Section Object
 - Provided by Win32 API
 - 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.
 - , thread critical section
 - Local object
 - kernel object가 , handle

2005-10-19

Seong Jong Choi

Multithreading_CHAPTER_4-7

4.2 Critical Sections

- Define critical section object
 - **CRITICAL_SECTION** critical_section;
- Initialized critical section
 - VOID **InitializeCriticalSection** (LPCRITICAL_SECTION lpCriticalSection);
- Deleting critical section object
 - VOID **DeleteCriticalSection** (LPCRITICAL_SECTION lpCriticalSection);

2005-10-19

Seong Jong Choi

Multithreading_CHAPTER_4-8

4.2 Critical Sections

- Entering the critical section
 - VOID `EnterCriticalSection` (LPCRITICAL_SECTION lpCriticalSection);
- Leaving the critical section
 - 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);
}
```

2005-10-19

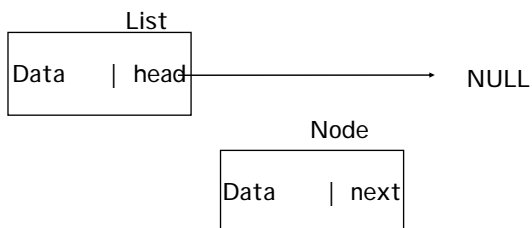
Seong Jong Choi

Multithreading_CHAPTER_4-11

4.2 Critical Sections

- Example : Linked list (P.79)

AddHead()



2005-10-19

Seong Jong Choi

Multithreading_CHAPTER_4-12

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;
}
```

2005-10-19

Seong Jong Choi

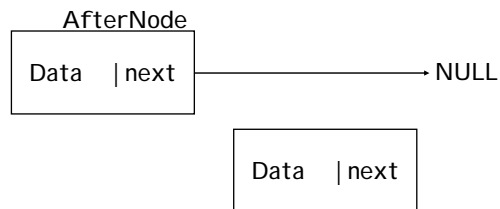
Multithreading_CHAPTER_4-13

4.2 Critical Sections

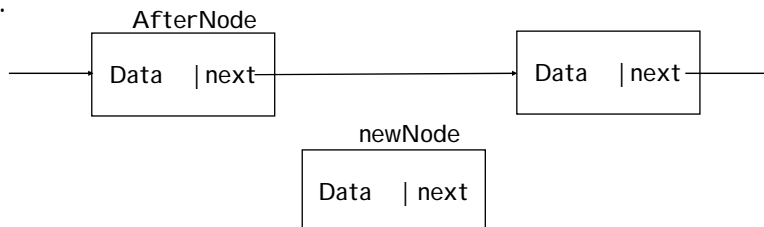
- Example : Linked list (P.79)

Insert()

1.



2.



2005-10-19

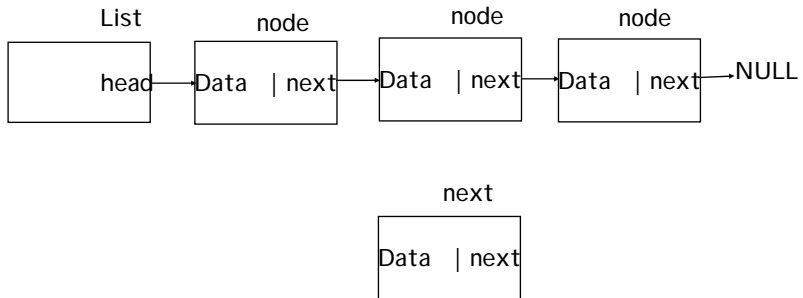
Seong Jong Choi

Multithreading_CHAPTER_4-14

4.2 Critical Sections

- Example : Linked list (P.79)

*Next()



Block

- - Critical Section
 - EnterCriticalSection(pCS)
 - EnterCriticalSection(pCS)
 - ...
 - LeaveCriticalSection(pCS)
 - LeaveCriticalSection(pCS)

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.

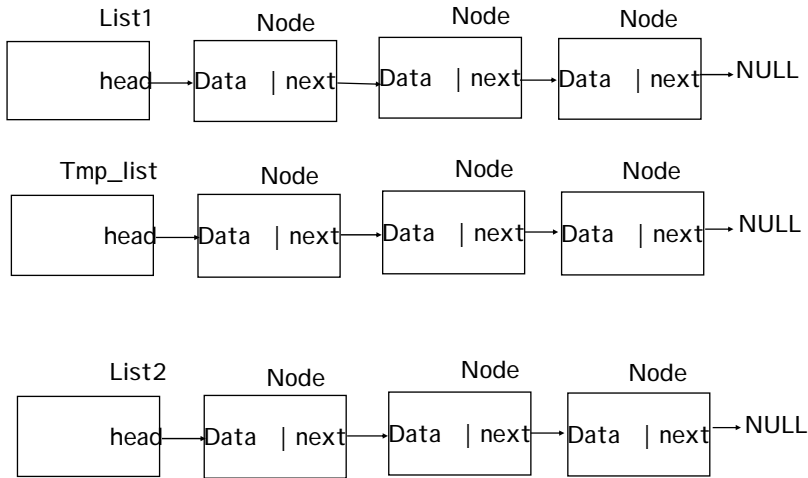
4.3 Deadlock

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



2005-10-19

Seong Jong Choi

Multithreading_CHAPTER_4-19

4.3 Deadlock

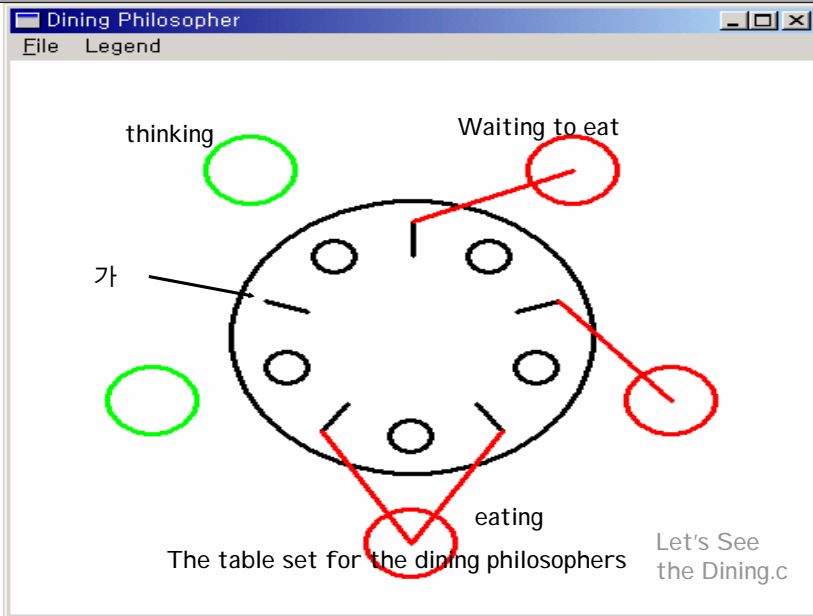
- Deadlock() = "The Deadly Embrace"
- Deadlock will not happen by allocating all the resources you need as a single operation with `WaitForMultipleObjects()`. (later...-> P.91)
- All-or-nothing proposition

2005-10-19

Seong Jong Choi

Multithreading_CHAPTER_4-20

4.4 The Dining Philosophers



2005-10-19

Seong Jong Choi

Multithreading_CHAPTER_4-21

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

2005-10-19

Seong Jong Choi

Multithreading_CHAPTER_4-22

4.5 Mutexes

- The comparison of the function

CRI TICAL_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` `blnHeritHandle`, // inheritance option

`LPCTSTR` `lpName` // object name);

- `dwDesiredAccess` : Specifies the requested access to the mutex.
- `blnHeritHandle` : 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);
```

- **Block**
 - Critical Section
 - EnterCriticalSection(pCS)
 - EnterCriticalSection(pCS)
 - ...
 - LeaveCriticalSection(pCS)
 - LeaveCriticalSection(pCS)
 - Mutex
 - WaitForSingleObject(hMtx)
 - WaitForSingleObject(hMtx)
 - ...
 - ReleaseMutex(hMtx)
 - ReleaseMutex(hMtx)
- Cf) Semaphore

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 lpAttribute,  
    LONG InitialCount,  
    LONG MaximumCount,  
    LPCTSTR lpName  
);
```

- lpAttribute: Use NULL to get the default Security Attributes.
- InitialCount: Specifies an initial count for the semaphore object ($0 \leq \text{this} \leq \text{MaximumCount}$)
- MaximumCount: Specifies the maximum count for the semaphore object
- lpName: 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,      // T=AutoReset, F=Manual Reset  
    BOOL bInitialState,    // initial state; T=signaled, F=nonsignaled  
    LPCTSTR lpName         // object name  
);
```

SetEvent()

Memory Operation

Set the event object to the signaled state

ResetEvent()

Set the event object to the nonsignaled state

PulseEvent()

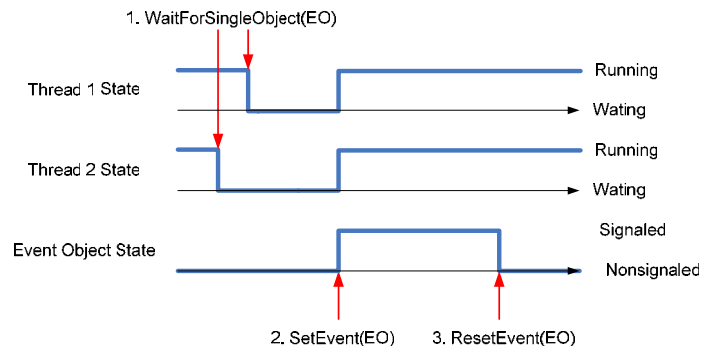
- Memoryless operation

- **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.

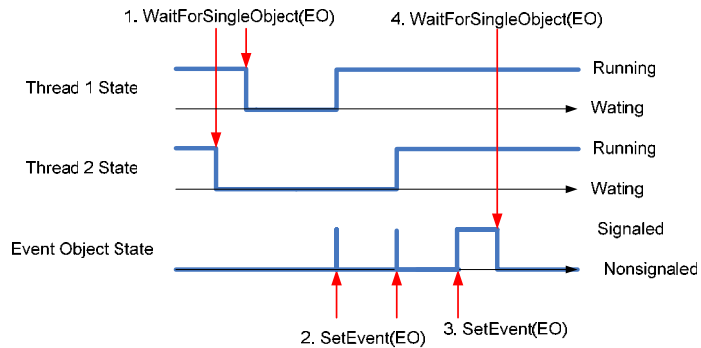
Manual Reset Event Object: Set/Reset

- Two threads are waiting for an event object (EO).
- SetEvent(EO) will activate **all threads** waiting for the EO.
- The EO will be in nonsignaled state only after the ResetEvent()



Auto Reset Event Object: Set

1. Two threads are waiting for an event object (EO).
2. SetEvent(EO) will activate **only one thread** waiting for the EO, then return to nonsignaled state
3. **If no threads are waiting, the event object's state remains signaled,**
4. **Until wait (or reset()) function.**



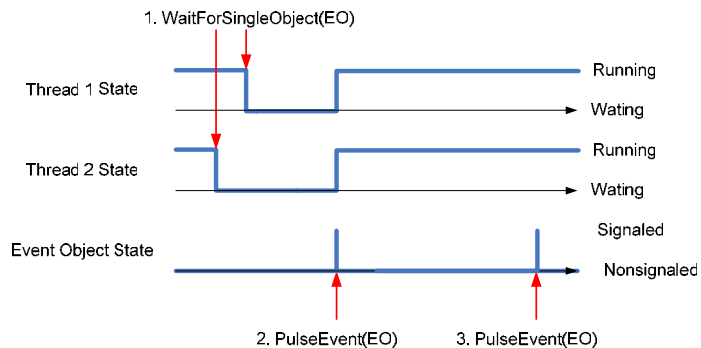
2005-10-19

Seong Jong Choi

Multithreading_CHAPTER_4-39

Manual Reset Event Object: Pulse

1. Assume two threads are waiting for an event object (EO).
2. PulseEvent(EO) will **activate all threads** waiting for the EO, then the EO return to the nonsignaled state.
3. If no threads are waiting, the event object's state remains nonsignaled



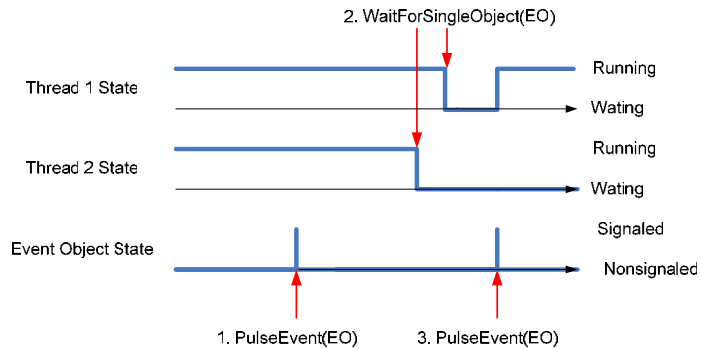
2005-10-19

Seong Jong Choi

Multithreading_CHAPTER_4-40

Auto Reset Event Object: Pulse

1. If no threads are waiting, the event object's state remains nonsignaled
2. Two threads are waiting for an event object (EO).
3. PulseEvent(EO) will **activate a single thread** waiting for the EO, then the EO return to the nonsignaled state.



2005-10-19

Seong Jong Choi

Multithreading_CHAPTER_4-41

4.7 Event Objects: Summary

- If threads are waiting.

	SetEvent()	ResetEvent()	PulseEvent()
Manual Reset	Activate all , then stay signaled	Nonsignaled	Activate all , then nonsignaled
Auto Reset	Activate one , then stay nonsignaled	---	Activate one , then nonsignaled

- If no threads are waiting.

	SetEvent()	ResetEvent()	PulseEvent()
Manual Reset	Stay signaled	Nonsignaled	Stay nonsignaled
Auto Reset	Stay signaled, until wait function.	---	Stay nonsignaled

2005-10-19

Seong Jong Choi

Multithreading_CHAPTER_4-42

Summary & Questions

- PulseEvent() is an memoryless operation.
- SetEvent() is memory operation.
- Manual/auto Reset
 - No threads are waiting, and current state is signaled.
What happens when PulseEvent()??

4.8 Displaying Output from Worker Threads

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

4.10 Summary of Synchronization Mechanisms

- Semaphore
 - Is a kernel object
 - Has no owner
 - Is Named
 - 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