# 멀티 쓰레드

HCI Programming 2 (321190)
2008년 가을학기
12/9/2008
박경신

## Overview

- 쓰레드의 개념과 동작 원리
- MFC 쓰레드의 두 종류인 작업자 쓰레드와 UI 쓰레드 사용법
- 다양한 쓰레드 동기화 기법을 이해하고 적용



## 멀티 태스킹과 멀티 쓰레딩

- 멀티 태스킹과 멀티 쓰레딩
  - 멀티 태스킹
    - 하나의 CPU가 여러 개의 프로세스를 교대로 수행
  - 멀티 쓰레딩
    - 하나의 CPU가 여러 개의 쓰레드를 교대로 수행
- 멀티 쓰레딩의 중요성
  - 응용 프로그램이 직접 쓰레드 생성과 파괴를 관리
  - 쓰레드 사용 여부에 따라 응용 프로그램의 성능 차이가 생기므로 중요한 프로그래밍 요소임

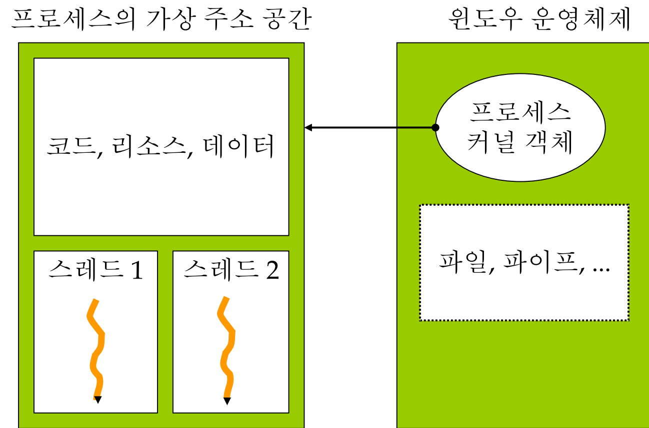


## 프로세스와 쓰레드

- 프로세스 (Process)
  - 실행 중인 프로그램
- 프로세스 구성 요소
  - 가상 주소 공간 - 32비트 윈도우의 경우 4기가 바이트
  - 가상 주소 공간에 로드된 실행 파일과 DLL(코드+리소스+데이터)
  - 프로세스를 위해 운영체제가 할당한 각종 리소스(파일, 파이프, ...)
  - 프로세스 커널 객체
  - 하나 이상의 스레드

## 프로세스와 쓰레드

- 프로세스 구성 요소





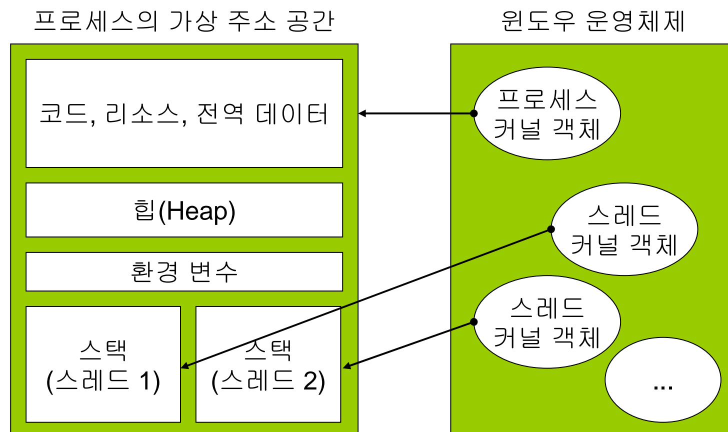

## 프로세스와 쓰레드

- 쓰레드 (Thread)
  - 프로세스의 가상 주소 공간에 존재하는 실행 흐름
  - 운영 체제는 각 쓰레드에게 일정한 CPU 시간을 교대로 할당함으로써 여러 개의 쓰레드가 병렬적으로 실행되는 효과를 만들어 냄
- 쓰레드 구성 요소
  - 스택
    - 커널 모드와 사용자 모드에서 실행하기 위한 두 개의 스택
  - 쓰레드 커널 객체
    - CPU 레지스터 값, ...



## 프로세스와 쓰레드

- 프로세스와 쓰레드 구성 요소






## 동기 (Motivation)

- 한 개의 응용프로그램이 비슷한 일을 여러 개를 동시에 수행해야 하는 경우가 있음
  - 여러 개의 클라이언트가 연결된 웹 서버
  - 동시에 여러 클라이언트의 요구를 받는 RPC (Remote Procedure Call) 서버
- 이럴 때마다 매번 프로세스를 만드는 작업은 시간을 많이 요구하고 힘듦
- 많은 운영체제의 커널들이 이미 멀티 쓰레드를 지원함; 커널에서도 이미 수많은 쓰레드들이 동작하고 있음
  - 각 쓰레드는 장치 또는 인터럽트 처리 등의 특정 작업을 수행함

## 동기 (Motivation)

- 프로세스 – 차지하는 기억 장소 + 사용하는 자원
- 문맥 교환 (Context Switching)
  - 한 프로세스에서 다른 프로세스로 제어가 넘어가면 이들 자원이 바뀌므로 다시 초기화
  - 레지스터, 스택, 프로그램 계수기 및 디스크 파일 등은 문맥 전환에 용이
  - 메모리 및 일부 주변 장치는 문맥 교환에 과다한 시간 소비
- 문맥 교환이 용이하지 않은 자원들을 공유하도록 하는 프로세스 개념이 바로 쓰레드임

## 장점 (Benefits)

- 응답성 (Responsiveness)
  - 인터랙티브 프로그램에서 일부분이 멈추거나 긴 작업을 수행하더라도 프로그램의 응답성을 높여줌
- 자원 공유 (Resource Sharing)
  - 쓰레드가 속한 프로세스의 메모리나 자원을 공유
- 경제성 (Economy)
  - 프로세스를 생성하는데 필요한 메모리나 자원을 사용 안 함
  - Sun Solaris의 경우에는 프로세스를 만드는 데 약 30배, 컨텍스트 스위칭에 5배 정도 느림
- 다중 CPU구조의 활용 (Utilization of MP Architectures)
  - 쓰레드는 다른 프로세서에서 실행될 수 있음

## 커널 쓰레드와 사용자 쓰레드

- 커널 쓰레드(Kernel Threads)
  - 커널이 직접 지원
  - Examples
    - Windows XP/2000
    - Sun Solaris
    - Linux
    - Tru64 UNIX
    - Mac OS X
- 사용자 쓰레드 (User Threads)
  - 사용자 수준의 라이브러리 형태로 지원
  - 커널이 관여하지 않음
  - 커널 수준 쓰레드보다 컨텍스트 스위칭이 빠름
  - 스케줄링이 공평치 못하게 진행되거나 전체 프로세스가 대기하는 문제 발생 가능

## 커널 쓰레드와 사용자 쓰레드

- 쓰레드 라이브러리
  - 쓰레드를 생성하고 관리하는 API 제공
  - 사용자와 커널 수준의 라이브러리로 구현 가능
- 많이 사용되는 쓰레드 라이브러리
  - POSIX pthreads
    - *커널 또는 사용자 쓰레드로 구현 가능*
    - POSIX standard (IEEE 1003.1c) 가 쓰레드의 생성과 동기화를 위해 제정한 API
    - API는 명세만 정의할 뿐이고, 실제 구현은 라이브러리 개발자가 정함
    - 유닉스 운영체제에서 공통적으로 제공됨 (Solaris, Linux, Mac OS X)
  - Win32 threads
    - 커널 수준
  - Java threads
    - 가상 기계의 구현 방법에 따름

## Windows CPU 스케줄링

- CPU 스케줄링 (CPU Scheduling)
  - 한정된 CPU 시간을 여러 쓰레드 (혹은 프로세스)로 분배하는 정책
  - 스케줄링은 사용자 프로세스와 입출력 시스템 호출을 포함한 시스템 프로세스에서 다룸
  - CPU가 유휴(idle) 상태가 될 때마다, 운영체제는 준비 큐에 있는 프로세스들 중에서 하나를 실행
- 윈도우의 CPU 스케줄링
  - 우선순위(Priority)에 기반한 CPU 스케줄링 기법을 사용
    - 우선순위가 높은 쓰레드에게 우선적으로 CPU 시간 할당
- 쓰레드의 우선순위 결정 요소
  - 프로세스 우선순위: 우선순위 클래스(Priority Class)
  - 쓰레드 우선순위: 우선순위 레벨(Priority Level)



## Windows CPU 스케줄링

- 우선순위 클래스
  - Windows는 프로세스들이 속할 수 있는 몇 가지 우선순위 클래스를 제공함
  - 하나의 프로세스가 생성한 쓰레드는 모두 동일한 우선순위 클래스를 가짐
- 우선순위 클래스 종류

REALTIME_PRIORITY_CLASS(실시간)
HIGH_PRIORITY_CLASS(높음)
ABOVE_NORMAL_PRIORITY_CLASS(보통 초과; 윈도우2000/XP)
NORMAL_PRIORITY_CLASS(보통)
BELOW_NORMAL_PRIORITY_CLASS(보통 미만; 윈도우2000/XP)
IDLE_PRIORITY_CLASS(낮음)



## Windows CPU 스케줄링

- 우선순위 클래스 종류





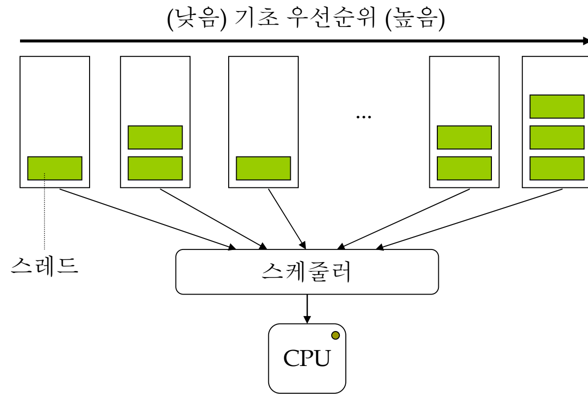
## Windows CPU 스케줄링

- 우선순위 레벨
  - 우선순위 클래스 안에 다시 상대적인 우선순위가 있음
  - 같은 프로세스에 속한 쓰레드 사이에서 상대적인 우선순위를 결정할 때 사용
- 우선순위 레벨 종류

THREAD_PRIORITY_TIME_CRITICAL
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_IDLE

# Windows CPU 스케줄링

- 우선순위 클래스 + 우선순위 레벨
  ⇨ 기초 우선순위(Base Priority)를 CPU 스케줄링에 사용






# Windows CPU 스케줄링

- Windows XP의 우선순위
  - 각 쓰레드의 우선 순위는 그 쓰레드가 속한 우선순위 클래스와 그 클래스 안에서의 상대적인 우선순위 레벨에 기반을 둠

| | real-time | high | above normal | normal | below normal | idle priority |
|---|---|---|---|---|---|---|
| time-critical | 31 | 15 | 15 | 15 | 15 | 15 |
| highest | 26 | 15 | 12 | 10 | 8 | 6 |
| above normal | 25 | 14 | 11 | 9 | 7 | 5 |
| normal | 24 | 13 | 10 | 8 | 6 | 4 |
| below normal | 23 | 12 | 9 | 7 | 5 | 3 |
| lowest | 22 | 11 | 8 | 6 | 4 | 2 |
| idle | 16 | 1 | 1 | 1 | 1 | 1 |



# Windows CPU 스케줄링

- 우선 순위 스케줄링 문제점
  - 기아 상태 (Starvation) : 낮은 우선 순위 작업들이 CPU를 사용 못하는 경우가 생김
- Windows의 CPU 스케줄링
  - 쓰레드의 시간 할당량이 만료되면, 인터럽트 당함
  - 인터럽트 된 가변 우선순위 쓰레드는 우선순위가 낮아짐
  - 가변 우선순위 쓰레드가 Wait상태에서 풀려나면, 디스패처는 우선순위를 높여줌. 단 얼마나 높여주느냐는 그 쓰레드가 무엇때문에 기다려 왔느냐에 달려 있음
    - 예) 키보드 입/출력을 기다렸다면 많이 높여주고, 디스크를 기다린 경우에는 보통으로 올림
  - Windows XP는 프로세스가 활성화되어 있으면 시간 할당량을 후위 프로세스에 비해 3배정도 할당함

# MFC 쓰레드

- MFC 쓰레드 종류
  - 작업자 쓰레드 (Worker Thread)
    - 메시지 루프가 없으므로 사용자 입력을 필요로 하지 않는 작업을 백그라운드로 수행할 때 사용
  - 사용자 인터페이스 쓰레드 (User Interface Thread)
    - 메시지 루프가 있으므로 사용자 입력을 받아서 처리할 때 사용
    - 윈도우95에서 folder를 생성하면 새로운 UI Thread가 생성
- MFC 쓰레드 생성
  - CWinThread 객체 생성 후 CreateThread() 호출
  - AfxBeginThread() 호출
- MFC 쓰레드 종료
  - AfxEndThread() 호출
  - Win32 함수 WaitForSingleObject/WaitForMultipleObject 사용

## 작업자 쓰레드

- 작업을 수행할 전역함수 선언

```
UINT ThreadProc (LPVOID param)          // 사용자 정의 함수
{ … }
```

- 쓰레드 생성
  - CWinThread 타입 객체를 동적으로 생성하고 (쓰레드를 만든 후) 이 객체의 주소값을 리턴

```
CWinThread* AfxBeginThread (
    AFX_THREADPROC pfnThreadProc,       // 함수이름
    LPVOID pParam,                      // 전달 파라메터
    int nPriority = THREAD_PRIORITY_NORMAL,
    UINT nStackSize = 0,
    DWORD dwCreateFlags = 0,
    LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL
);
```



## 작업자 쓰레드

- 쓰레드 생성
  - pfnThreadProc: 쓰레드 실행 시작점이 되는 함수(=제어 함수)의 주소
    - 제어 함수의 형태 ⇨ **UINT 함수이름 (LPVOID pParam);**
  - pParam: 제어 함수에 전달할 인자(32비트)
  - nPriority: 쓰레드의 우선순위 레벨
  - nStackSize: 쓰레드 스택의 크기
  - dwCreateFlags: 0 또는 CREATE_SUSPENDED
  - lpSecurityAttrs: 보안 설명자와 핸들 상속 정보



## 작업자 쓰레드

- 쓰레드에 데이터 넘기기
  - 전역변수 이용
  - LParam 이용

```
UINT ThreadProc (LPVOID LParam)
{
    CMExThreadView *pView = (CMExThreadView *)lParam;
    // 스레드 내용…
}
```



## 작업자 쓰레드

- 쓰레드 제어
  - 쓰레드 우선순위 레벨 값을 얻음

```
int CWinThread::GetThreadPriority ();
```

  - 쓰레드 우선순위 레벨 값을 변경

```
BOOL CWinThread::SetThreadPriority (int nPriority);
```

  - 쓰레드 실행을 일시 정지

```
DWORD CWinThread::SuspendThread ();
```

  - 쓰레드를 재시작

```
DWORD CWinThread::ResumeThread ();
```

# 작업자 쓰레드

- 쓰레드 종료
  1. 쓰레드 제어 함수가 리턴. 리턴 값이 0이면 일반적으로 정상 종료를 뜻함
  2. 쓰레드 제어 함수 내에서 AfxEndThread() 호출



# 작업자 쓰레드

- 작업자 쓰레드 종료

```
void AFXAPI AfxEndThread(
   UINT nExitCode,
   BOOL bDelete = TRUE
);
```

  - nExitCode: 쓰레드 종료 코드
  - bDelete: 쓰레드 객체를 메모리에서 제거할 것인지를 나타냄. FALSE를 사용하면 쓰레드 객체 재사용 가능



# UI 쓰레드

- UI 쓰레드 생성 과정
  - CWinThread 클래스로부터 새로운 클래스를 파생.
  - 클래스 선언부와 구현부에 각각 DECLARE_DYNCREATE, IMPLEMENT_DYNCREATE 매크로 선언.
  - CWinThread 클래스가 제공하는 가상 함수 중 일부를 재정의. CWinThread::InitInstance()는 반드시 재정의해야 하며, 나머지 함수는 필요에 따라 재정의함.
  - AfxBeginThread()를 이용하여 새로운 UI 쓰레드 생성.



# UI 쓰레드

- 쓰레드 생성
  - CWinThread 타입 객체를 동적으로 생성하고 (쓰레드를 만든 후) 이 객체의 주소값을 리턴

```
CWinThread* AfxBeginThread (
   CRuntimeClass* pThreadClass,
   int nPriority = THREAD_PRIORITY_NORMAL,
   UINT nStackSize = 0,
   DWORD dwCreateFlags = 0,
   LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL
);
```

## UI 쓰레드

- 쓰레드 생성
  - pThreadClass: 객체 정보를 담고 있는 CRuntimeClass 구조체의 주소
    - 사용 형태 ⇨ **RUNTIME_CLASS(클래스이름)**
  - nPriority: 쓰레드의 우선순위 레벨
  - nStackSize: 쓰레드 스택의 크기
  - dwCreateFlags: 0 또는 CREATE_SUSPENDED
  - lpSecurityAttrs: 보안 설명자와 핸들 상속 정보



## UI 쓰레드

- 쓰레드 종료
  1. WM_QUIT 메시지를 받아서 메시지 루프가 종료
  2. 쓰레드 제어 함수 내에서 AfxEndThread() 호출



## UI 쓰레드

- UI thread 생성

```
// The CUIThread class
class CUIThread : public CWinThread
{
DECLARE_DYNCREATE(CUIThread)
public:
   virtual BOOL InitInstance();
};
IMPLEMENT_DYNCREATE(CUIThread, CWinThread)
BOOL CUIThread::InitInstance()
{
   m_pMainWnd = new CMainWindow;
   m_pMainWnd->ShowWindow(SW_SHOW);
   m_pMainWnd->UpdateWindow();
   return TRUE;
}
```

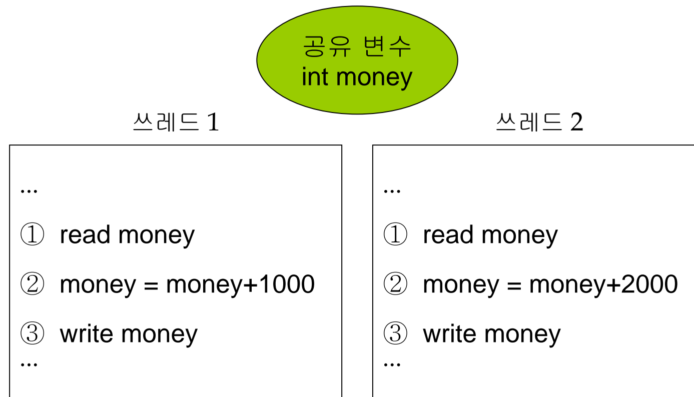
```
// The CMainWindow class
class CMainWindow : public CFrameWnd
{
public:
   CMainWindow();
protected:
   afx_msg void OnLButtonDown(UINT, CPoint);
   DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(CMainWindow, CFrameWnd)
   ON_WM_LBUTTONDOWN()
END_MESSAGE_MAP()

CMainWindow::CMainWindow()
{
   Create(NULL, “UI Thread Window”);
}
void CMainWindow::OnLButtonDown(UINT nFlags, CPoint point)
{
   PostMessage(WM_CLOSE, 0, 0);
}
CWinThread* pThread = AfxBeginThread(RUNTIME_CLASS(CUIThread));
```

# 쓰레드 동기화 (Thread Synchronization)

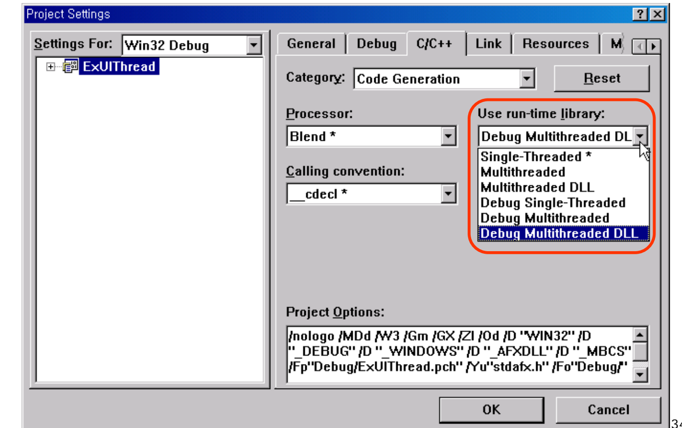
- 쓰레드 동기화(Thread Synchronization)가 필요한 상황






# 쓰레드 동기화 (Thread Synchronization)

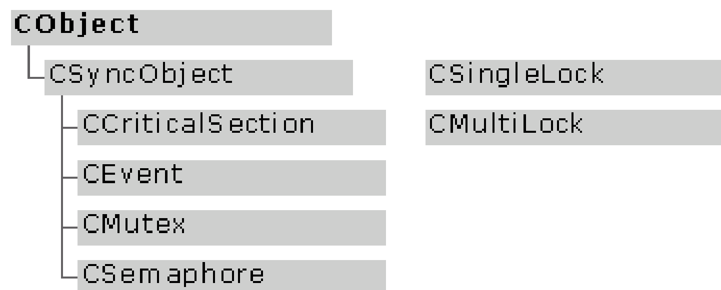
- C/C++ 라이브러리 선택






# 쓰레드 동기화 (Thread Synchronization)

- MFC 클래스 계층도



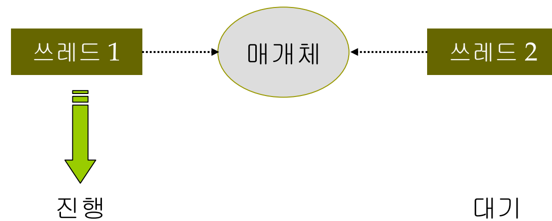



# 쓰레드 동기화 (Thread Synchronization)

- 클래스 요약
  - CSyncObject
    - 쓰레드 동기화 클래스를 위한 공통의 인터페이스 제공
  - CCriticalSection, CMutex, Csemaphore, CEvent
    - 윈도우 운영체제에서 제공하는 **쓰레드 동기화 객체(임계 영역, 뮤텍스, 세마포, 이벤트)**를 편리하고 일관성 있게 사용할 수 있도록 만든 **클래스**
    - **임계영역을 제외한 나머지는 kernel object**
  - CSingleLock, CMultiLock
    - 쓰레드 동기화 클래스를 편리하게 사용할 수 있도록 보조하는 **동기화 접근을 허용하는 클래스**
- #include <afxmt.h> 필요

# 쓰레드 동기화 (Thread Synchronization)

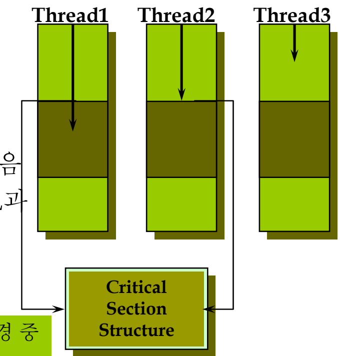
- 쓰레드 동기화가 필요한 상황
  - 두 개 이상의 쓰레드가 공유 리소스를 사용하는 경우
  - 하나의 쓰레드가 작업을 완료한 후, 기다리고 있던 다른 모든 쓰레드에게 알려주는 경우
- 쓰레드 동기화 원리






# 임계 영역 (Critical Section)

- 용도
  - **같은 프로세스에 속한 쓰레드 간의 동기화**
  - 공유 리소스를 접근하는 다수의 쓰레드가 있을 때 오직 하나의 쓰레드만 접근할 수 있도록 함
- 장점
  - 속도가 빠름
- 단점
  - 서로 다른 프로세스에 속한 쓰레드 간의 동기화를 위한 목적으로는 사용할 수 없음
  - 여러 프로세스가 동시에 접근하는 DLL과 같은 파일에서는 사용할 수 없음




-Thread1은 임계영역안에 있고, Critical Section Structure를 변경 중
-Thread2가 임계영역에 도착했으나 Thread1때문에 못 들어감
-Thread3은 아직 임계영역에 도착하지 못함

# 임계 영역 (Critical Section)

- 사용 예

```
// 전역 변수로 선언
CCriticalSection g_cs;
...
// 쓰레드 1
UINT Thread1(LPVOID lParam) {
  g_cs.Lock();
  // 공유 변수 접근
  g_cs.Unlock();
}
// 쓰레드 2
UINT Thread2(LPVOID lParam) {
  g_cs.Lock();
  // 공유 변수 접근
  g_cs.Unlock();
}
```



# 뮤텍스 (Mutex)

- 커널 객체로서 Mutual Exclusion의 약자
- 임계영역(Critical Section)과 유사
  - 여러 개의 쓰레드가 한 개의 뮤텍스를 공유하고 이것을 가지고 있는 한 개의 쓰레드만 작업
- 용도
  - **공유 리소스를 접근하는 다수의 쓰레드가 있을 때 오직 하나의 쓰레드만 접근할 수 있도록 함**
- 장점
  - **서로 다른 프로세스(Multi-process)에 속한 쓰레드 간의 동기화를 위한 목적으로 사용할 수 있음**
  - 프로세스간 동기화 가능함
- 단점
  - 임계 영역보다 속도가 느림

# 뮤텍스 (Mutex)

## 뮤텍스 생성

```
CMutex::CMutex (
    BOOL bInitiallyOwn = FALSE,
    LPCTSTR lpszName = NULL,
    LPSECURITY_ATTRIBUTES lpsaAttribute = NULL
);
```

- bInitiallyOwn: TRUE면 뮤텍스를 생성한 쓰레드가 소유자가 됨
- lpszName: 뮤텍스 이름
- lpsaAttribute: 보안 설명자와 핸들 상속 관련 구조체



# 뮤텍스 (Mutex)

## 사용 예

```
// 전역 변수로 선언
CMutex g_mutex(FALSE, NULL);
...
// 쓰레드 1
UINT Thread1(LPVOID lParam) {
    g_mutex.Lock();
    // 공유 변수 접근
    g_mutex.Unlock();
}
// 쓰레드 2
UINT Thread2(LPVOID lParam) {
    g_mutex.Lock();
    // 공유 변수 접근
    g_mutex.Unlock();
}
```



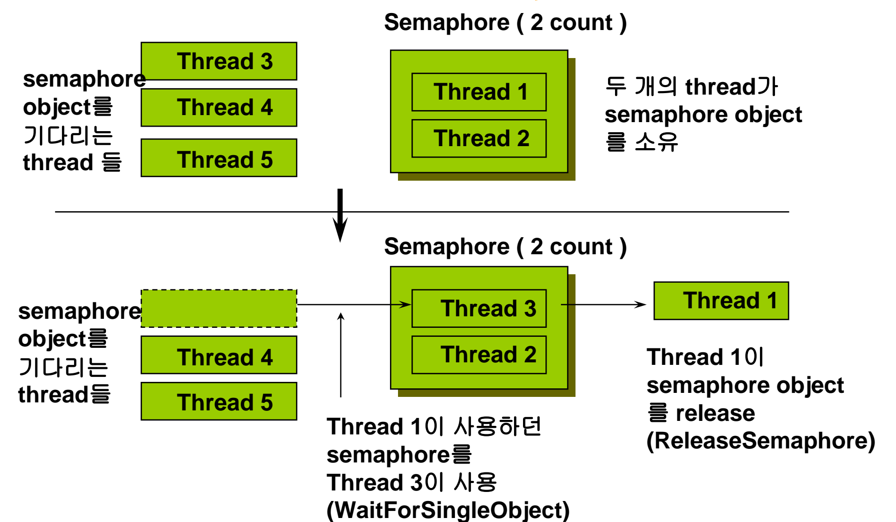
# 세마포 (Semaphore)

## 세마포

- 임계영역과 뮤텍스는 오직 한 개의 쓰레드만 동작하는 반면 세마포는 **다수의 원하는 개수의 쓰레드 만큼 동시에 작업할 수 있도록 함**
- 한정된 개수의 자원을 여러 쓰레드가 접근하려고 할 때, 이를 제어하는 동기화 객체
- 사용 가능한 리소스의 개수(=**리소스 카운트**)를 유지하여 수행될 수 있는 쓰레드 개수를 조절
- 생성시 쓰레드의 번호와 최대 허용 가능한 쓰레드의 개수를 정의함



# 세마포 (Semaphore)



# 세마포 (Semaphore)

- 세마포를 이용한 동기화
  - 세마포를 생성. 이때 사용 가능한 자원의 개수로 **리소스 카운트를 초기화**
  - 리소스를 사용할 쓰레드는 자신이 필요한 리소스 개수만큼 Lock()을 호출하며, Lock()이 성공할 때마다 리소스 카운트 값이 1씩 감소. 리소스 카운트가 0인 상태에서 Lock()을 호출하면 해당 쓰레드는 대기함.
  - 리소스 사용을 마친 쓰레드는 자신이 사용한 리소스 개수만큼 Unlock()을 호출하며, 이때마다 리소스 카운트 값이 1씩 증가



# 세마포 (Semaphore)

- 세마포 생성

```
CSemaphore::CSemaphore (
   LONG lInitialCount = 1,
   LONG lMaxCount = 1,
   LPCTSTR pstrName = NULL,
   LPSECURITY_ATTRIBUTES lpsaAttributes = NULL
);
```

  - lInitialCount: 세마포의 초기값
  - lMaxCount: 세마포의 최대값
  - pstrName: 세마포 이름
  - lpsaAttribute: 보안 설명자와 핸들 상속 관련 구조체

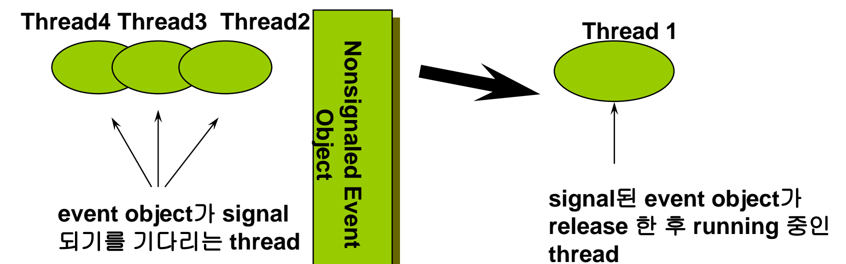


# 이벤트 (Event)

- 이벤트
  - 동기화 객체 중 가장 기본적인 형태
  - 임계영역, 뮤텍스, 세마포는 데이터 접근을 제어하기 위해 사용하나, 이벤트는 어떤 동작이 완료되었음에 대한 신호에 사용
  - 신호(Signaled)와 비신호(Nonsignaled) 두 개의 상태를 가진 동기화 객체
- 용도
  - 두 개 이상의 쓰레드가 공유 리소스를 사용하는 경우
    ⇨ 임계 영역, 뮤텍스, **이벤트(자동 리셋 Auto-Reset)**
  - 하나의 쓰레드가 작업을 완료한 후, 기다리고 있던 다른 모든 쓰레드에게 알려주는 경우
    ⇨ **이벤트(수동 리셋 Manual-Reset)**



# 이벤트 (Event)



# 이벤트 (Event)

- 이벤트 객체를 이용한 동기화
  - 이벤트 객체를 비신호 상태로 생성
  - 하나의 쓰레드가 초기화 작업을 진행하고, 나머지 쓰레드는 이벤트 객체에 대해 Lock()을 호출함으로써 이벤트 객체가 신호 상태가 되기를 기다림
  - 쓰레드가 초기화 작업을 완료하면 이벤트 객체를 신호 상태로 바꿈
  - 기다리고 있던 모든 쓰레드가 깨어나서 작업을 진행



# 이벤트 (Event)

- 종류
  - 자동 리셋 (Auto Reset)
    - 수동 리셋 이벤트보다는 뮤텍스나 세마포와 비슷
    - Signal되면 wait 하는 쓰레드를 깨운 후 resume 하기 직전에 자동으로 이벤트 객체의 상태를 nonsignaled state로 reset
    - 기다리던 쓰레들 중에 어떤 것을 resume할 지 정할 수 없음. High priority thread가 run
    - 이벤트 객체를 신호 상태로 바꾸면, 기다리는 쓰레드 중 하나만 깨운 후 자동으로 비신호 상태가 됨
  - 수동 리셋 (Manual Reset)
    - WaitFor .. 함수가 nonsignaled 상태로 자동으로 reset하지 않음
    - 이벤트 객체를 신호 상태로 바꾸면, 계속 신호 상태를 유지. 결과적으로 기다리는 쓰레드를 모두 깨우게 됨.
    - 리셋을 하려면 명시적으로 함수를 호출해야 함



# 이벤트 (Event)

- 이벤트 생성

```
CEvent::CEvent (
   BOOL bInitiallyOwn = FALSE,
   BOOL bManualReset = FALSE,
   LPCTSTR lpszName = NULL,
   LPSECURITY_ATTRIBUTES lpsaAttribute = NULL
);
```

  - bInitiallyOwn: FALSE면 비신호, TRUE면 신호 상태
  - bManualReset: FALSE면 자동 리셋, TRUE면 수동 리셋
  - lpszName: 이벤트 이름
  - lpsaAttribute: 보안 설명자와 핸들 상속 관련 구조체



# 이벤트 (Event)

- 이벤트 상태 변경
  - 이벤트 객체를 신호 상태로 바꿈

```
BOOL CEvent::SetEvent();
```

  - 이벤트 객체를 비신호 상태로 바꿈

```
BOOL CEvent::ResetEvent();
```

  - 이벤트 객체를 신호 상태로 바꾸고, 신호 상태를 기다리는 다른 쓰레드를 깨운 후, 다시 비신호 상태로 바꿈

```
BOOL CEvent::PulseEvent();
```

    - 수동 리셋 이벤트인 경우, 기다리던 모든 쓰레드를 release한 후 비신호 상태로 바꿈
    - 자동 리셋 이벤트인 경우, 단 한 개의 쓰레드만 깨움

# 사용할 동기화 클래스 선택요령

- 프로그램이 어떤 리소스에 접근하기 위해서 다른 일이 발생하기를 기다릴 필요가 있는가 ?
  - ➔ CEvent 사용
- 동시에 여러 개의 클래스가 한 개의 리소스에 접근할 필요가 있는가 ?
  - ➔ CSemaphore 사용
- 한 개 이상의 프로그램이 리소스에 접근하는가 ?
  - ➔ CMutex 사용



# CSingleLock 클래스

- 문제 발생

```
CMutex g_mutex(...);

MyThread()
{
    g_mutex.Lock();

    // 예외 상황 발생 -> return

    g_mutex.Unlock();
}
```



# CSingleLock 클래스

- 해결 방법

```
CMutex g_mutex(...);

MyThread()
{
    CSingleLock lock(&g_mutex);

    lock.Lock();

    // 예외 상황 발생

    lock.Unlock();
}
```



# CMultiLock 클래스

- 사용 예1

```
CEvent g_event[3];
CSyncObject* g_pSyncObjects[3] = {
    &g_event[0], &g_event[1], &g_event[2]
};

MyThread()
{
    CMultiLock multiLock(g_pSyncObjects, 3);
    multiLock.Lock();
    ...
}
```

# CMultiLock 클래스

- 사용 예2

```
CEvent g_event[3];
CSyncObject* g_pSyncObjects[3] = {
    &g_event[0], &g_event[1], &g_event[2]
};

MyThread()
{
    CMultiLock multiLock(g_pSyncObjects, 3);
    multiLock.Lock(INFINITE, FALSE);
    ...
}
```