

# 멀티쓰레드 프로그래밍과 동기화 객체

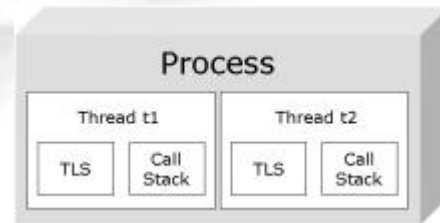
김성익 (noerror@hitel.net)  
2005.03.31

# 도입

- **멀티 테스킹, 멀티 쓰레드, 멀티 프로세서**
- **멀티 쓰레드 프로그래밍 활용**
- **싱글 쓰레드와 멀티 쓰레드의 차이**
- **동기화 객체**

# 멀티쓰레드

- 프로세스(Process)란 ?



- 스레드(Thread)란 ?

- **컨텍스트 스위칭 (Context Switching)**  
쓰레드 레벨의 컨텍스트 스위칭, 프로세스 레벨의 컨텍스트 스위칭 차이  
컨텍스트 스위칭 비용, 메모리 공유

# 멀티쓰레드 프로그래밍

- 다수의 쓰레드를 동시에 활용하여 개발
- 병목 현상을 줄이고 효율을 높인다.  
컨텍스트 스위칭의 비용에도 불구하고 더 빠른 효율을 가지도록 접근
- 개별적인 작업을 동시에 한다.  
연관성이 없는 작업을 어렵게 끼워 맞출 필요가 없음

# 병목

- IO등의 작업으로 블록 상태로 전이할 경우

- 예제  
극단적인 경우지만 Sleep 에 해당하는 작업 동안에는 CPU는 대기  
결과값은 약 2000.

```
#include <stdio.h>
#include <windows.h>

void sum10()
{
    int i, total=0;
    for(i=0; i<10; i++)
    {
        total += i;
        Sleep(100);
    }
}

void sum100()
{
    int i, total=0;
    for(i=0; i<100; i++)
    {
        total += i;
        Sleep(10);
    }
}

void main()
{
    int pivot = timeGetTime();
    sum10();
    sum100();
    printf("%d\n", timeGetTime() - pivot);
}
```

# 병목 제거

- 멀티 스레드로 개선  
결과는 약1000.

```
#include <stdio.h>
#include <windows.h>

DWORD WINAPI sum10(void *)
{
    int i, total=0;
    for(i=0; i<10; i++)
    {
        total += i;
        Sleep(100);
    }
    return 0;
}

DWORD WINAPI sum100(void *)
{
    int i, total=0;
    for(i=0; i<100; i++)
    {
        total += i;
        Sleep(10);
    }
    return 0;
}
```

```
void main()
{
    DWORD threadid;
    HANDLE handle[2];
    int pivot = timeGetTime();
    handle[0] = CreateThread(NULL, 0, sum10, NULL, 0, &threadid);
    handle[1] = CreateThread(NULL, 0, sum100, NULL, 0, &threadid);
    WaitForMultipleObjects(2, handle, TRUE, INFINITE);
    printf("%d\n", timeGetTime() - pivot);
}
```

# 사용 / 관련 함수

- **CreateThread**  
쓰레드를 생성 (beginthread과 같은 기능)  
주의) 마지막 인자 NULL인 경우 98에서는 Fail
- **TerminateThread**  
쓰레드를 강제로 정지  
사용 지양 (자바는 초기 시절에만 존재)
- **SetThreadPriority/GetThreadPriority**
- **WaitForSingleObject/MultipleObjects**
- **CloseHandle**

# 컨텍스트 스위칭 조건

- 블럭모드로 진입(Sleep 함수, IO 작업)  
*sleep(0) ??*  
스케줄링 룰에 따라 (임의제어불가)

# 동기화 문제 (1)

- 멀티 쓰레드는 메모리 공유

```
#include <stdio.h>
#include <windows.h>

static volatile int g_ShareMemory = 0;

DWORD WINAPI func1(void *)
{
    while(1)
    {
        g_ShareMemory = 1;
        if (g_ShareMemory != 1)
            printf("ERROR\n");
        g_ShareMemory = 0;
    }

    return 0;
}

void main()
{
    DWORD threadid;

    CreateThread(NULL, 0, func1, NULL, 0, &threadid);
    CreateThread(NULL, 0, func1, NULL, 0, &threadid);
    CreateThread(NULL, 0, func1, NULL, 0, &threadid);

    Sleep(INFINITE);
}
```

# 동기화 문제(2)

- 스레드만 봤을 때는 논리적 오류가 없지만 컨텍스트 스위칭 일어난 시점에 따라 치명적인 문제 발생

```
9:      while(1)
00401038  mov     eax,1
0040103D  test   eax,eax
0040103F  je     func1+56h (00401076)
10:    {
11:      if (g_ShareMemory == 0)
00401041  cmp    dword ptr [g_ShareMemory (00427c48)],0
00401048  jne    func1+54h (00401074)
12:    {
13:      g_ShareMemory = 1;
0040104A  mov    dword ptr [g_ShareMemory (00427c48)],1
14:      if (g_ShareMemory != 1)
00401054  cmp    dword ptr [g_ShareMemory (00427c48)],1
0040105B  je     func1+4Ah (0040106a)
15:      printf("ERROR#n");
0040105D  push  offset string "ERROR#n" (0042201c)
00401062  call  printf (00401180)
00401067  add   esp,4
16:      g_ShareMemory = 0;
0040106A  mov    dword ptr [g_ShareMemory (00427c48)],0
17:    }
18:  }
00401074  jmp   func1+18h (00401038)
```

# 동기화 문제(3)

- 레지스터에 저장된 이전 값을 사용하는 경우

```
17:
➔ 0040D7C5  mov     eax,[g_ShareMemory (00427e80)]
   0040D7CA  add     eax,1
   0040D7CD  mov     [g_ShareMemory (00427e80)],eax
```

# 동기화 처리

- 동기화 객체이용

```
#include <stdio.h>
#include <windows.h>

static CRITICAL_SECTION g_CS;
static int g_ShareMemory = 0;

DWORD WINAPI func1(void *)
{
    while(1)
    {
        EnterCriticalSection(&g_CS);

        if (g_ShareMemory == 0)
        {
            g_ShareMemory = 1;
            if (g_ShareMemory != 1)
                printf("ERROR\n");
            g_ShareMemory = 0;
        }

        LeaveCriticalSection(&g_CS);
    }

    return 0;
}
```

```
void main()
{
    DWORD threadid;

    InitializeCriticalSection(&g_CS);

    CreateThread(NULL, 0, func1, NULL, 0, &threadid);
    CreateThread(NULL, 0, func1, NULL, 0, &threadid);
    CreateThread(NULL, 0, func1, NULL, 0, &threadid);

    Sleep(INFINITE);
}
```

# 동기화 객체

- 기능 : Enter/Leave
- **크리티컬 섹션** *Critical Section*
- **뮤텍스** *Mutex*
- **세마포어** *Semaphore*
- **이벤트** *Event*

# 크리티컬 섹션(1)

- **하나의 쓰레드만 통과 가능**  
프로세스(process)간의 동기화에 사용 불가
- **다른 쓰레드가 사용중이면 블럭됨**  
CPU를 다른 쓰레드로 넘김
- **커널 레벨의 객체가 아니라 빠르다**  
뮤텍스, 세마포어등의 다른 객체에 비해서
- **한 쓰레드에서 여러 번 호출시 무시**  
데드락을 막기 위함 ???

# 크리티컬 섹션(2)

- **사용**

```
static CRITICAL_SECTION g_CS;

InitializeCriticalSection(&g_CS); // 생성
...
EnterCriticalSection(&g_CS); // Enter
// Thread Safe한 작업 가능
LeaveCriticalSection(&g_CS); // Leave
....
DeleteCriticalSection(&g_CS); // 소멸
```

- **스핀락 *spinlock***

멀티 프로세서 환경에서 빈번한 컨텍스트 스위칭을 막기 위해 바로 스위칭 하지 않고 일정 카운트 만큼 CPU가 대기

# 세마포어 (1)

- N개만큼 통과 가능
- Named Semaphore는 프로세스간의 동기화 처리도 가능
- 쓰레드, 프로세스 무관하게 카운트  
한 쓰레드에서 모두 소유한 상태에서 다시 Enter 하면 데드락 (비교. 크리티컬섹션)

# 세마포어 (2)

- **사용흐름**

```
HANDLE g_hSemap;  
g_hSemap = CreateSemaphore(NULL, N, N, NULL); // 생성  
...  
WaitForSingleObject(g_hSemap, INFINITE); // Enter  
// N개 이내 통과  
ReleaseSemaphore(g_hSemap, 1, NULL); // Leave  
....  
CloseHandle(g_hSemap); // 소멸
```

- 초기 N에서 사용시 -1, 사용 후 +1 =>  
카운트가 0이면 대기 상태로

# 뮤텍스

- 카운트가 1인 세마 포어
- 사용흐름

```
HANDLE g_hMutex;  
g_hMutex = CreateMutex(NULL, FALSE, NULL); // 생성  
  
WaitForSingleObject(g_hSemap, INFINITE); // Enter  
// 단독으로 통과  
ReleaseMutex(g_hMutex); // Leave  
CloseHandle(g_hSemap); // 소멸
```

# 이벤트(1)

- **시그널을 제어 할 수 있다**  
근본적으로 Mutex, Semaphore와 다른점  
Mutex, Semaphore는 wait and signal이 무조건 자동, 이벤트는 수동 Reset이 가능하다
- **IO 동기화 등에 사용됨**  
시그널을 함수로 셋 가능한 특성 이용

# 이벤트(2)

- 수동 리셋 / 자동 리셋 방식
- 자동시그널 사용흐름 (mutex 유사)

```
HANDLE hEvent;  
  
hEvent = CreateEvent(NULL, FALSE, TRUE, NULL); // 생성, 대기후 자동 년시그널로, 초기 시그널  
WaitForSingleObject(hEvent, INFINITE); // 년시그널이면 대기, 년시그널로  
// SAFE  
SetEvent(hEvent); // 시그널  
CloseHandle(hEvent);
```

- 수동시그널  
흐름을 외부에서 제어하는 데 주로 사용

# 데드락

- 동기화 객체에서 대기하지만 영원히 대기하게 되는 경우  
같은 쓰레드에서 사용중인 동기화 객체를 다시 통과하려고 할 때 영원히 통과할 수 없는 경우  
서로 다른 쓰레드에서 상대 객체를 대기하면서 기다리는 경우
- 논리적 오류

# 응용

- N개의 read 허용하며, write시는 하나만 가능하고, 이때 read 는 불가능

```
// READ
// LOCK
HANDLE hList[2] = { g_hSemap, g_hMutex };
WaitForMultipleObjects(sizeof(hList)/sizeof(*hList), hList, TRUE, INFINITE);
ReleaseMutex(g_hMutex);

// READ
// UNLOCK
ReleaseSemaphore(g_hSemap, 1, &prevcnt);
```

```
// WRITE
// LOCK
WaitForSingleObject(g_hMutex, INFINITE);
for(int i=0; i<_MAXCNT; i++)
    WaitForSingleObject(g_hSemap, INFINITE);

// WRITE
// UNLOCK
ReleaseSemaphore(g_hSemap, _MAXCNT, NULL);
ReleaseMutex(g_hMutex);
```

# 결론

- 멀티 쓰레드에서의 동기화의 까다로움
- 동기화의 문제는 쉽게 발견되지 않음
- 그럼에도 불구하고 효율을 높일 수 있다면?