Ch 07 Synchronization Examples

Objectives

- 유한 버퍼, readers-writer 및 식사하는 철학자 동기화 문제에 관해 설명한다.
- 프로세스 동기화 문제를 해결하기 위해 Linux 및 Windows에서 사용하는 특정 도구를 설명한다.
- 프로세스 동기화 문제를 해결하기 위하여 POSIX 및 Java가 어떻게 사용될 수 있는지 설명한다.

동기화에 대한 전통적인 문제 3가지

Bounded-Buffer Problem(Producer-Consumer Problem)

제한된 버퍼에 데이터를 채우고/가져가는 문제

소비자와 생산자가 공유하는 자료구조

```
int n;
//mutex 이진 세마포는 버퍼 풀에 접근하기 위한 상호 배제 기능을 제공하며 1
semaphore mutex = 1;
//empty와 full 세마포들은 각각 비어 있는 버퍼의 수와 꽉 찬 버퍼의 수를 ㄱ
semaphore empty = n;
semaphore full = 0;
```

생산자와 소비자 프로세스의 구조

🌟 생산자와 소비자 코드의 대칭성에 주목할 것, 생산자가 소비자를 위해 꽉 찬 버퍼를 생산해내고, 소비자는 생산자를 위해 비어 있는 버퍼를 생산해내는 것으로 해석

```
while(true) {
    ...
    /*produce an item in next_produced*/
    ...
    //만약 만약 버퍼가 꽉 차있다면, 생산자는 여기에 대기
    //empty 세마포어의 값이 0보다 크면,
    //생산자는 즉시 다음 단계로 진행할 수 있고, 그렇지 않으면 공간이 생길
    wait(empty);
    //mutex=1이면 임계구역 사용가능, 0이면 대기!
    wait(mutex);
    //mutex가 0이 아니라면 버퍼에서 다음 값을 생산
    ...
    /*add next_produced to the buffer*/
    ...
    //버퍼에 대한 접근이 끝났음을 알리고,
    //다른 프로세스가 버퍼에 접근할 수 있도록 mutex 세마포어를 해제
    signal(mutex);
    //버퍼에 새로운 아이템이 추가되었음을 알림
    signal(full);
}
```

전체 코드 구조

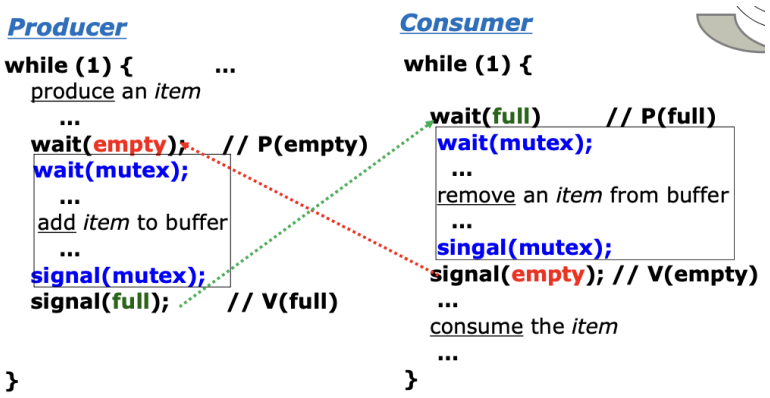
1. 무한 루프를 통해 지속적으로 아이템을 생산합니다.
2. 새로운 아이템을 생성하여 `next_produced`에 저장합니다.
3. `empty` 세마포어를 기다려서 버퍼에 빈 공간이 있는지 확인합니다.
4. `mutex` 세마포어를 기다려서 버퍼에 대한 접근 권한을 획득합니다.
5. `next_produced`를 버퍼에 추가합니다.
6. `mutex` 세마포어를 해제하여 버퍼에 대한 접근을 다른 프로세스가 할 수 있도록 합니다.
7. `full` 세마포어를 신호하여 버퍼에 새로운 아이템이 추가되었음을 알립니다.

```
while(true) {
    //full 값이 0이면 대기(소비할 자원이 없음)
    wait(full);
    //mutex 값이 0이면 대기(임계구역 진입 불가)
    wait(mutex);
```

```
    //mutex 값이 1이면 버퍼 진입, 자원 소비
    ...
    /*remove an item from buffer to next_consumed*/
    ...
    //버퍼에 대한 접근이 끝났음을 알리고, 다른 프로세스가 버퍼에 접근할 수
    //mutex 세마포어 해제(mutex=1)
    signal(mutex);
    //버퍼가 비었음을 알림
    signal(empty);
    ...
    /*consume the item in next_consumed*/
}
```

전체 코드 구조

1. 무한 루프를 통해 지속적으로 아이템을 소비합니다.
2. `full` 세마포어를 기다려서 꽉 찬 버퍼가 있는지 확인합니다.
3. `mutex` 세마포어를 기다려서 버퍼에 대한 접근 권한을 획득합니다.
4. 버퍼의 next_consumed를 소비합니다.
5. `mutex` 세마포어를 해제하여 버퍼에 대한 접근을 다른 프로세스가 할 수 있도록 합니다.
6. `empty` 세마포어를 신호하여 버퍼가 소비되었음을 알립니다.



Readers-Writers Problem

- 하나의 데이터베이스가 다수의 병행 프로세스 간에 공유된다고 가정할 때, Readers 프로세스는 데이터베이스의 내용을 읽기만 하고 Writers 프로세스는 데이터를 갱신(읽고 쓰기) 할 수 있다.

- 만약 두 reader가 동시에 공유 데이터에 접근하더라도, 문제가 발생하지는 않지만 하나의 writer와 어떤 다른 스레드(reader 혹은 writer)가 동시에 데이터베이스에 접근하면, 혼란이 야기될 수 있다.
- 이러한 문제점이 발생하지 않도록 보장하기 위해, 우리는 writer가 쓰기 작업을 하는 동안에 공유 데이터베이스에 대해 배타적 접근 권한을 가지게 할 필요가 있다. 이 동기화 문제를 readers-writers 문제라고 한다.
- readers-writers 문제에는 여러 가지 변형들이 있는데, 모두 우선순위와 연관된 변형들이다.

공유 데이터

```
//writer의 상호배제를 보장(1이면 writer가 데이터 갱신 가능)
semaphore rw_mutex = 1;
//read_count를 갱신할 때 상호배제를 보장
semaphore mutex = 1;
//객체를 읽고 있는 프로세스의 개수
int read_count = 0;
```

reader와 writer 프로세스의 구조

```
while(true) {
    //rw_mutex값이 1이 되어 임계구역 진입이 가능할 때까지 대기
    wait(rw_mutex);
    ...
    /* writing is performed */
    ...
    //rw_mutex값 다시 1로 변경(다른 프로세스의 진입 허용)
    signal(rw_mutex);
}
```

```
while(true) {
    //read_count 변경 임계구역에 진입이 가능할 때(mutex=1)까지 대기
    wait(mutex);
    read_count++;
    //만약 객체를 읽고 있는 프로세스가 있다면 rw_mutex=1이 될 때까지 대
    if(read_count == 1)
        wait(rw_mutex);
```

```
    //read_count 값 변경이 가능함을 알림
    signal(mutex);
    ...
    /*reading is performed*/
    ...
    //read_count 변경 임계구역에 진입이 가능할 때(mutex=1)까지 대기
    wait(mutex);
    read_count--;
    //현재 읽고 있는 프로세스가 없다면 rw_mutex 사용이 가능함을 알림
    if(read_count == 0)
        signal(rw_mutex);
    //mutex 임계구역 사용이 가능함(read_count값 변경 가능)을 알림
    signal(mutex);
}
```

Dining-Philosophers Problem Algorithm

문제 상황

- 5명의 철학자가 있다고 할 때, 철학자들은 원형 테이블을 공유하며, 이 테이블은 각각 한 철학자에 속하는 5개의 의자로 둘러싸여 있다. 테이블 중앙에는 한 사발의 밥이 있고, 테이블에는 다섯 개의 젓가락이 놓여 있다. 철학자가 생각할 때는 다른 동료들과 상호 작용하지 않는다.
- 때때로 철학자들은 배가 고파지고 자신에게 가장 가까이 있는 두 개의 젓가락(자신과 자신의 왼쪽 철학자 그리고 오른쪽 철학자 사이에 있는 젓가락)을 집으려고 시도한다. 철학자는 한 번에 한 개의 젓가락만 집을 수도 있다. 배고픈 철학자가 동시에 젓가락 두 개를 집으면, 젓가락을 놓지 않고 식사를 쉰다. 식사를 마치면, 젓가락 두 개를 모두 놓고 다시 생각하기 시작한다.

세마포 해결안

- 한 가지 간단한 해결책은 각 젓가락을 하나의 세마포로 표현하는 것이다. 철학자는 그 세마포에 wait() 연산을 실행하여 젓가락을 집으려고 시도한다. 또한 해당 세마포에 signal() 연산을 실행함으로써 자신의 젓가락을 놓는다.
- 공유 자료는 다음과 같다. `semaphore chopstick[5];`

철학자 i의 구조

```
while(true){
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5);
    ...
    /*eat for a while*/
    ...
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5]);
    ...
    /*think for while*/
    ...
}
```

Windows API - PART 2(Process/Thread Synchronization)

Windows 동기화 객체 종류

- **스레드간** 동기화 객체 – **유저 모드 동기화 기법**
  - CRITICAL_SECTION
    - 동일 프로세스내에 포함된 **여러 개의 스레드가 공유 자원에 접근할 때의 배타적인 제어**
    - **프로세스간 동기화에는 사용할 수 없음**
- **프로세스/스레드간** 동기화 객체 – **커널 모드 동기화 기법**

동기화 객체	용도
Mutex	리소스의 배타적 제어
Semaphore	리소스를 동시에 사용할 수 있는 프로세스/스레드 수 제어
Event	다른 스레드에 이벤트 통지하여 객체의 시그널/비시그널 상태를 자유롭게 제어

객체의 시그널/비시그널 상태

- 대기용 API – 단일/복수 객체가 시그널 상태가 되는 것을 대기
  - DWORD WaitForSingleObject(// 반환값:처리 결과
    HANDLE hHandle, // 객체의 핸들
    DWORD dwMilliseconds) // 최대대기시간(msec)
  - DWORD WaitForMultipleObjects(// 반환값:처리 결과
    DWORD nCount, // 객체 수
    CONST HANDLE lpHandles, // 객체 핸들의 배열
    BOOL bWaitAll, // 전부 시그널상태 ? TRUE-all
    DWORD dwMilliSeconds) // 최대대기시간(msec)

- 객체의 시그널/비시그널 상태

객체 종류	시그널 상태	비시그널 상태
프로세스/스레드	실행을 마치면	실행중
뮤텍스	어느 스레드/프로세스에도 소유되지 않을 때	어느 스레드/프로세스에 소유되어 있을 때
세마포	관리하는 카운터값이 1 이상	관리하는 카운터값이 0일 때

Semaphores

- Semaphore *S* : integer variable
- Can only be accessed via **two indivisible (atomic) operations**
  - Originally called **P(or wait)** and **V(or signal)**

```
wait (S):  while (S≤ 0) do  no-op;
           S--;
```

i.e. wait

If positive, decrement-&-enter.
Otherwise, wait until it gets positive

```
signal (S):
           S++;
```

Semaphore 객체

- Semaphore 관련 API
  - HANDLE **CreateSemaphore**(// 반환값: 세마포어 객체 핸들
    LPSECURITY_ATTRIBUTES lpSemAttributes, // 보안속성
    LONG InitialCount, // 카운트 초기값
    LONG IMaximumCount, // 카운트 최대값
    LPCTSTR lpName) // 세마포어 이름 문자열
  - HANDLE **ReleaseSemaphore**(// 반환값: 처리 결과
    HANDLE hSemaphore, // 세마포어 핸들
    LONG IReleaseCount, // 해제할 카운트 수
    LPLONG lpPreviousCount)// 호출 전 카운트 수

  - **대기용 API(WaitForSingleObject())**는 시그널 상태에서 실행을 재개하는 동시에 세마포어 카운트가 1씩 감소한다. **Cf) wait()**
  - **ReleaseSemaphore()** API로 세마포어 카운트가 증가. **Cf) signal()**

Event 객체(1)

- 객체의 시그널 상태를 프로그램에서 자유롭게 제어
- 용어

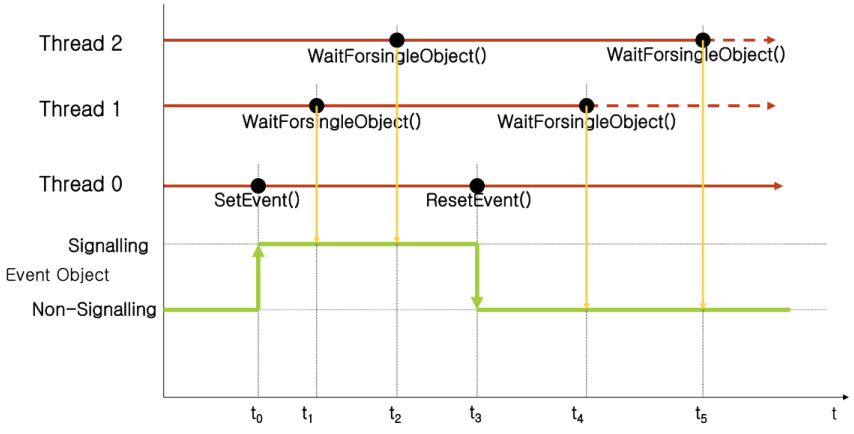
- 이벤트 관련 API
  - HANDLE CreateEvent(// 반환값: 이벤트 객체 핸들
    LPSECURITY_ATTRIBUTES lpEventAttributes, // 보안속성
    BOOL bManualReset, // 수동/자동 리셋 설정–TRUE:수동,FALSE:자동
    BOOL bInitialState, // 이벤트의 초기 시그널 상태
    LPCTSTR lpName)// 이벤트 이름 문자열
  - HANDLE SetEvent(HANDLE hEvent)
  - HANDLE ResetEvent(HANDLE hEvent)
  - HANDLE PulseEvent(HANDLE hEvent)

Event 객체(2)

리셋형	SetEvent() API	PulseEvent() API
수동 리셋형	ResetEvent() API가 호출될 때까지 시그널 상태를 지속한다	이미 대기중인 모든 스레드의 실행을 재개 한 후 자동으로 리셋한다
자동 리셋형	대기중인 스레드를 하나만 실행 재개하고 자동으로 리셋한다	이미 대기중인 한 스레드의 실행을 재개한 후 자동으로 리셋한다. 대기중인 스레드가 없으면 곧바로 리셋한다

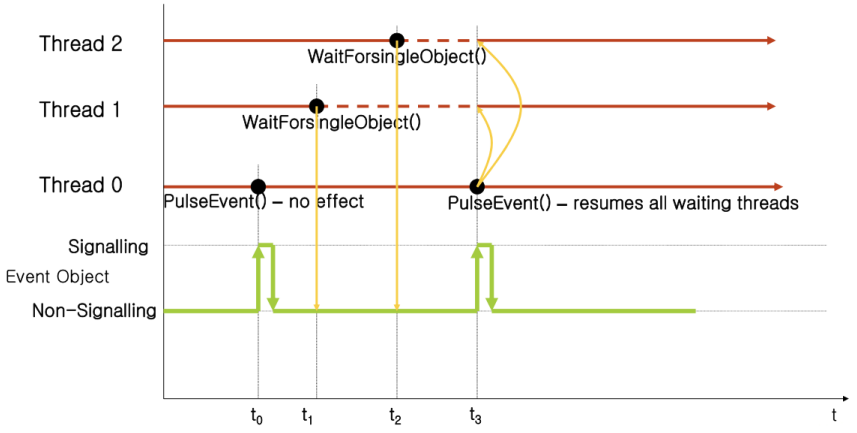
Event 객체(3)

- 수동 Reset형 – SetEvent() API



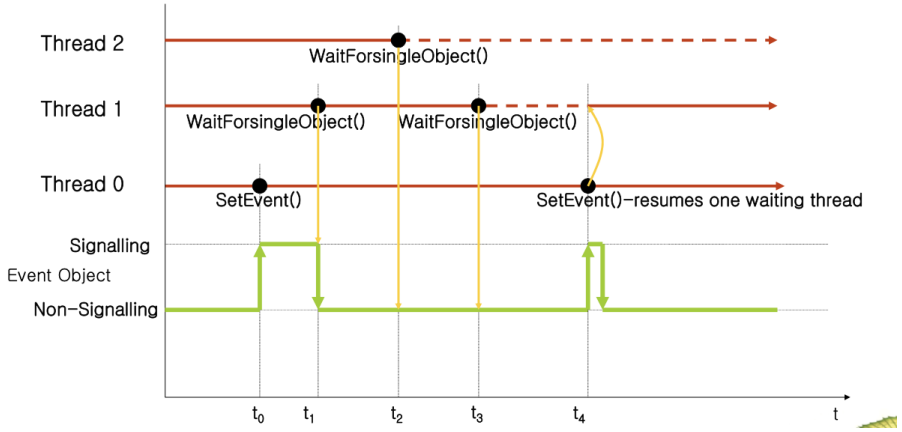
Event 객체(4)

- 수동 Reset형 – PulseEvent() API



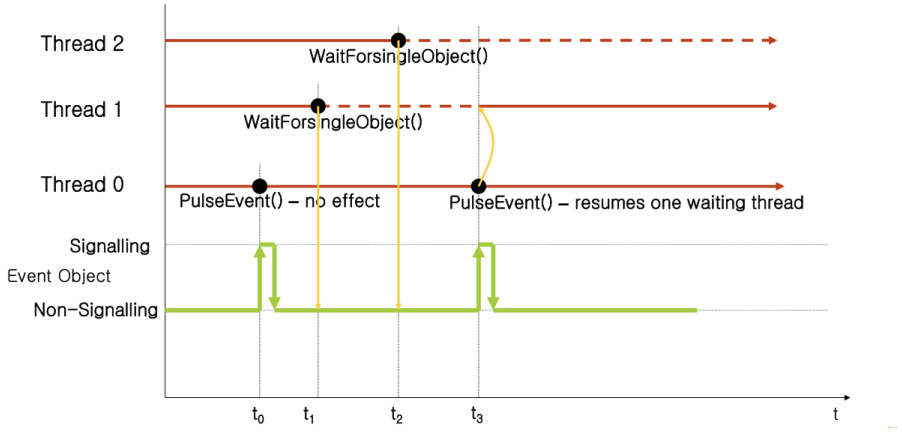
Event 객체(5)

- 자동 Reset형 – SetEvent() API



Event 객체(6)

■ 자동 Reset형 – PulseEvent() API



Examples in Synchronization of Cooperation Processes(Threads)

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

int ncount; // 전역 변수, 생산자와 소비자가 공유하는 카운터

// 생산자 함수
void Producer(void* arg) {
    int i, mydata;
    for(i=0; i<10; i++) {
        mydata = ncount; // ncount 값을 mydata에 복사
        mydata++; // mydata 값을 증가
        Sleep(5); // 5 밀리초 동안 대기
        ncount = mydata; // 증가된 mydata 값을 ncount에 저장
        printf("Producer => %d\n", ncount); // 현재 ncount 값을
    }
}

// 소비자 함수
void Consumer(void* arg) {
    int i, mydata;
    for(i=0; i<10; i++) {
        mydata = ncount; // ncount 값을 mydata에 복사
```

```
        mydata++; // mydata 값을 증가
        Sleep(5); // 5 밀리초 동안 대기
        ncount = mydata; // 증가된 mydata 값을 ncount에 저장
        printf("Consumer => %d\n", ncount); // 현재 ncount 값을
    }
}

void main() {
    HANDLE hThreadVector[2]; // 스레드 핸들을 저장할 배열
    DWORD ThreadID; // 스레드 ID를 저장할 변수
    int i, mydata;

    ncount = 0; // 전역 변수 초기화

    // 생산자 스레드 생성
    hThreadVector[0] = CreateThread(
        NULL, // 기본 보안 속성
        0, // 기본 스택 크기
        (LPTHREAD_START_ROUTINE) Producer, // 실행할 함수
        NULL, // 함수에 전달할 매개변수
        0, // 기본 스레드 생성 플래그
        &ThreadID // 생성된 스레드의 ID를 저장할 변수
    );

    // 소비자 스레드 생성
    hThreadVector[1] = CreateThread(
        NULL, // 기본 보안 속성
        0, // 기본 스택 크기
        (LPTHREAD_START_ROUTINE) Consumer, // 실행할 함수
        NULL, // 함수에 전달할 매개변수
        0, // 기본 스레드 생성 플래그
        &ThreadID // 생성된 스레드의 ID를 저장할 변수
    );

    // 메인 스레드에서 ncount 증가 작업 수행
    for(i=0; i<10; i++) {
        mydata = ncount; // ncount 값을 mydata에 복사
        mydata++; // mydata 값을 증가
```

```
        Sleep(5); // 5 밀리초 동안 대기
        ncount = mydata; // 증가된 mydata 값을 ncount에 저장
        printf("Main => %d\n", ncount); // 현재 ncount 값을 출력
    }

    // 생산자 및 소비자 스레드가 종료될 때까지 대기
    WaitForMultipleObjects(
        2, // 대기할 스레드의 수
        hThreadVector, // 스레드 핸들의 배열
        TRUE, // 모든 스레드가 종료될 때까지 대기
        INFINITE // 무한 대기
    );

    // 최종 ncount 값을 출력
    printf("Final Value => %d\n", ncount);
}
```

문제점

- 이 코드는 여러 스레드가 동시에 `ncount`에 접근하고 수정하기 때문에 **경쟁 조건(Race Condition)** 문제가 발생할 수 있습니다. 이를 해결하려면 **상호 배제(Mutex)** 같은 동기화 메커니즘을 사용하여 각 스레드가 `ncount`에 접근할 때 동기화를 보장해야 합니다.

Critical Section을 이용하여 Race Condition 해결

- 추가된 코드(보라색)

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

CRITICAL_SECTION cs;
int ncount; // 전역 변수, 생산자와 소비자가 공유하는 카운터

// 생산자 함수
void Producer(void* arg) {
    int i, mydata;
    for(i=0; i<10; i++) {
        EnterCriticalSection( &cs );
            mydata = ncount; // ncount 값을 mydata에 복사
            mydata++; // mydata 값을 증가
```

```
            Sleep(5); // 5 밀리초 동안 대기
            ncount = mydata; // 증가된 mydata 값을 ncount에 저장
            printf("Producer => %d\n", ncount); // 현재 ncount
        LeaveCriticalSection( &cs );
    }
}

// 소비자 함수
void Consumer(void* arg) {
    int i, mydata;
    for(i=0; i<10; i++) {
        EnterCriticalSection( &cs );
            mydata = ncount; // ncount 값을 mydata에 복사
            mydata++; // mydata 값을 증가
            Sleep(5); // 5 밀리초 동안 대기
            ncount = mydata; // 증가된 mydata 값을 ncount에 저장
            printf("Consumer => %d\n", ncount); // 현재 ncount
        LeaveCriticalSection( &cs );
    }
}

void main() {
    HANDLE hThreadVector[2]; // 스레드 핸들을 저장할 배열
    DWORD ThreadID; // 스레드 ID를 저장할 변수
    int i, mydata;

    InitializeCriticalSection( &cs );
    ncount = 0; // 전역 변수 초기화

    // 생산자 스레드 생성
    hThreadVector[0] = CreateThread(
        NULL, // 기본 보안 속성
        0, // 기본 스택 크기
        (LPTHREAD_START_ROUTINE) Producer, // 실행할 함수
        NULL, // 함수에 전달할 매개변수
        0, // 기본 스레드 생성 플래그
        &ThreadID // 생성된 스레드의 ID를 저장할 변수
    );
```

```
    // 소비자 스레드 생성
    hThreadVector[1] = CreateThread(
        NULL, // 기본 보안 속성
        0, // 기본 스택 크기
        (LPTHREAD_START_ROUTINE) Consumer, // 실행할 함수
        NULL, // 함수에 전달할 매개변수
        0, // 기본 스레드 생성 플래그
        &ThreadID // 생성된 스레드의 ID를 저장할 변수
    );

    // 메인 스레드에서 ncount 증가 작업 수행
    for(i=0; i<10; i++) {
        EnterCriticalSection( &cs );
            mydata = ncount; // ncount 값을 mydata에 복사
            mydata++; // mydata 값을 증가
            Sleep(5); // 5 밀리초 동안 대기
            ncount = mydata; // 증가된 mydata 값을 ncount에 저장
            printf("Main => %d\n", ncount); // 현재 ncount 값을
        LeaveCriticalSection( &cs );
    }

    // 생산자 및 소비자 스레드가 종료될 때까지 대기
    WaitForMultipleObjects(
        2, // 대기할 스레드의 수
        hThreadVector, // 스레드 핸들의 배열
        TRUE, // 모든 스레드가 종료될 때까지 대기
        INFINITE // 무한 대기
    );
    DeleteCriticalSection( &cs );

    // 최종 ncount 값을 출력
    printf("Final Value => %d\n", ncount);
}
```

Mutex를 이용하여 Race Condition 해결

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

HANDLE hMutex;
int ncount; // 전역 변수, 생산자와 소비자가 공유하는 카운터

// 생산자 함수
void Producer(void* arg) {
    int i, mydata;
    for(i=0; i<10; i++) {
        WaitForSingleObject( hMutex, INFINITE );
            mydata = ncount; // ncount 값을 mydata에 복사
            mydata++; // mydata 값을 증가
            Sleep(5); // 5 밀리초 동안 대기
            ncount = mydata; // 증가된 mydata 값을 ncount에 저장
            printf("Producer => %d\n", ncount); // 현재 ncount
        ReleaseMutex( hMutex );
    }
}

// 소비자 함수
void Consumer(void* arg) {
    int i, mydata;
    for(i=0; i<10; i++) {
        WaitForSingleObject( hMutex, INFINITE );
            mydata = ncount; // ncount 값을 mydata에 복사
            mydata++; // mydata 값을 증가
            Sleep(5); // 5 밀리초 동안 대기
            ncount = mydata; // 증가된 mydata 값을 ncount에 저장
            printf("Consumer => %d\n", ncount); // 현재 ncount
        ReleaseMutex( hMutex );
    }
}

void main() {
    HANDLE hThreadVector[2]; // 스레드 핸들을 저장할 배열
    DWORD ThreadID; // 스레드 ID를 저장할 변수
    int i, mydata;
```

```
hMutex = CreateMutex(NULL, FALSE, "sample_mutex");
ncount = 0; // 전역 변수 초기화

// 생산자 스레드 생성
hThreadVector[0] = CreateThread(
    NULL, // 기본 보안 속성
    0, // 기본 스택 크기
    (LPTHREAD_START_ROUTINE) Producer, // 실행할 함수
    NULL, // 함수에 전달할 매개변수
    0, // 기본 스레드 생성 플래그
    &ThreadID // 생성된 스레드의 ID를 저장할 변수
);

// 소비자 스레드 생성
hThreadVector[1] = CreateThread(
    NULL, // 기본 보안 속성
    0, // 기본 스택 크기
    (LPTHREAD_START_ROUTINE) Consumer, // 실행할 함수
    NULL, // 함수에 전달할 매개변수
    0, // 기본 스레드 생성 플래그
    &ThreadID // 생성된 스레드의 ID를 저장할 변수
);

// 메인 스레드에서 ncount 증가 작업 수행
for(i=0; i<10; i++) {
    WaitForSingleObject( hMutex, INFINITE );
        mydata = ncount; // ncount 값을 mydata에 복사
        mydata++; // mydata 값을 증가
        Sleep(5); // 5 밀리초 동안 대기
        ncount = mydata; // 증가된 mydata 값을 ncount에 저장
        printf("Main => %d\n", ncount); // 현재 ncount 값을
    ReleaseMutex( hMutex );
}

// 생산자 및 소비자 스레드가 종료될 때까지 대기
WaitForMultipleObjects(
    2, // 대기할 스레드의 수
```

```
        hThreadVector, // 스레드 핸들의 배열
        TRUE, // 모든 스레드가 종료될 때까지 대기
        INFINITE // 무한 대기
    );
    CloseHandle( hMutex );

    // 최종 ncount 값을 출력
    printf("Final Value => %d\n", ncount);
}
```

세마포를 이용하여 Race Condition 해결

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

HANDLE hSemaphore;
int ncount; // 전역 변수, 생산자와 소비자가 공유하는 카운터

// 생산자 함수
void Producer(void* arg) {
    int i, mydata;
    for(i=0; i<10; i++) {
        WaitForSingleObject( hSemaphore, INFINITE );
            mydata = ncount; // ncount 값을 mydata에 복사
            mydata++; // mydata 값을 증가
            Sleep(5); // 5 밀리초 동안 대기
            ncount = mydata; // 증가된 mydata 값을 ncount에 저장
            printf("Producer => %d\n", ncount); // 현재 ncount
        ReleaseSemaphore( hSemaphore, 1, NULL );
    }
}

// 소비자 함수
void Consumer(void* arg) {
    int i, mydata;
    for(i=0; i<10; i++) {
        WaitForSingleObject( hSemaphore, INFINITE );
            mydata = ncount; // ncount 값을 mydata에 복사
```

```
            mydata++; // mydata 값을 증가
            Sleep(5); // 5 밀리초 동안 대기
            ncount = mydata; // 증가된 mydata 값을 ncount에 저장
            printf("Consumer => %d\n", ncount); // 현재 ncount
        ReleaseSemaphore( hSemaphore, 1, NULL );
    }
}

void main() {
    HANDLE hThreadVector[2]; // 스레드 핸들을 저장할 배열
    DWORD ThreadID; // 스레드 ID를 저장할 변수
    int i, mydata;

    hSemaphore = CreateSemaphore(NULL,1,1,"sample_semaphore")
    ncount = 0; // 전역 변수 초기화

    // 생산자 스레드 생성
    hThreadVector[0] = CreateThread(
        NULL, // 기본 보안 속성
        0, // 기본 스택 크기
        (LPTHREAD_START_ROUTINE) Producer, // 실행할 함수
        NULL, // 함수에 전달할 매개변수
        0, // 기본 스레드 생성 플래그
        &ThreadID // 생성된 스레드의 ID를 저장할 변수
    );

    // 소비자 스레드 생성
    hThreadVector[1] = CreateThread(
        NULL, // 기본 보안 속성
        0, // 기본 스택 크기
        (LPTHREAD_START_ROUTINE) Consumer, // 실행할 함수
        NULL, // 함수에 전달할 매개변수
        0, // 기본 스레드 생성 플래그
        &ThreadID // 생성된 스레드의 ID를 저장할 변수
    );

    // 메인 스레드에서 ncount 증가 작업 수행
    for(i=0; i<10; i++) {
```

```
        WaitForSingleObject( hSemaphore, INFINITE );
            mydata = ncount; // ncount 값을 mydata에 복사
            mydata++; // mydata 값을 증가
            Sleep(5); // 5 밀리초 동안 대기
            ncount = mydata; // 증가된 mydata 값을 ncount에 저장
            printf("Main => %d\n", ncount); // 현재 ncount 값을
        ReleaseSemaphore( hSemaphore, 1, NULL );
    }

    // 생산자 및 소비자 스레드가 종료될 때까지 대기
    WaitForMultipleObjects(
        2, // 대기할 스레드의 수
        hThreadVector, // 스레드 핸들의 배열
        TRUE, // 모든 스레드가 종료될 때까지 대기
        INFINITE // 무한 대기
    );
    CloseHandle( hSemaphore );

    // 최종 ncount 값을 출력
    printf("Final Value => %d\n", ncount);
}
```

이벤트를 이용하여 Race Condition 해결

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

HANDLE hEvent;
int ncount; // 전역 변수, 생산자와 소비자가 공유하는 카운터

// 생산자 함수
void Producer(void* arg) {
    int i, mydata;
    for(i=0; i<10; i++) {
        WaitForSingleObject( hEvent, INFINITE );
            mydata = ncount; // ncount 값을 mydata에 복사
            mydata++; // mydata 값을 증가
            Sleep(5); // 5 밀리초 동안 대기
```

```
            ncount = mydata; // 증가된 mydata 값을 ncount에 저장
            printf("Producer => %d\n", ncount); // 현재 ncount
        SetEvent( hEvent );
    }
}

// 소비자 함수
void Consumer(void* arg) {
    int i, mydata;
    for(i=0; i<10; i++) {
        WaitForSingleObject( hEvent, INFINITE );
            mydata = ncount; // ncount 값을 mydata에 복사
            mydata++; // mydata 값을 증가
            Sleep(5); // 5 밀리초 동안 대기
            ncount = mydata; // 증가된 mydata 값을 ncount에 저장
            printf("Consumer => %d\n", ncount); // 현재 ncount
        SetEvent( hEvent );
    }
}

void main() {
    HANDLE hThreadVector[2]; // 스레드 핸들을 저장할 배열
    DWORD ThreadID; // 스레드 ID를 저장할 변수
    int i, mydata;

    hEvent = CreateEvent(NULL,FALSE,TRUE,NULL);
    ncount = 0; // 전역 변수 초기화

    // 생산자 스레드 생성
    hThreadVector[0] = CreateThread(
        NULL, // 기본 보안 속성
        0, // 기본 스택 크기
        (LPTHREAD_START_ROUTINE) Producer, // 실행할 함수
        NULL, // 함수에 전달할 매개변수
        0, // 기본 스레드 생성 플래그
        &ThreadID // 생성된 스레드의 ID를 저장할 변수
    );
```

```
    // 소비자 스레드 생성
    hThreadVector[1] = CreateThread(
        NULL, // 기본 보안 속성
        0, // 기본 스택 크기
        (LPTHREAD_START_ROUTINE) Consumer, // 실행할 함수
        NULL, // 함수에 전달할 매개변수
        0, // 기본 스레드 생성 플래그
        &ThreadID // 생성된 스레드의 ID를 저장할 변수
    );

    // 메인 스레드에서 ncount 증가 작업 수행
    for(i=0; i<10; i++) {
        WaitForSingleObject( hEvent, INFINITE );
            mydata = ncount; // ncount 값을 mydata에 복사
            mydata++; // mydata 값을 증가
            Sleep(5); // 5 밀리초 동안 대기
            ncount = mydata; // 증가된 mydata 값을 ncount에 저장
            printf("Main => %d\n", ncount); // 현재 ncount 값을
        SetEvent( hEvent );
    }

    // 생산자 및 소비자 스레드가 종료될 때까지 대기
    WaitForMultipleObjects(
        2, // 대기할 스레드의 수
        hThreadVector, // 스레드 핸들의 배열
        TRUE, // 모든 스레드가 종료될 때까지 대기
        INFINITE // 무한 대기
    );
    CloseHandle( hEvent );

    // 최종 ncount 값을 출력
    printf("Final Value => %d\n", ncount);
}
```