중간고사 자료

다중 선택	멀티쓰레드
구성내용요약	범위 : 1~5주차
정리 완료	☐
파일	멀티쓰레드 프로그래밍.zip

커널
프로세스와 스레드
컨텍스트스위칭

스레스 생성 (CreateThread(), _beginThreadex())
WaitForSingleObject(), WaitForMultipleObjects()

경쟁조건
동기화도구 (Critical Section, Mutex, Semaphore, Event)

volatile
멀티스레드 환경 문제(경쟁조건 교착상태, 기아상태, 속도 불균형)
흐름제어 (생성자-소비자 패턴, 리더-팔로워 패턴, 파이프라인 처리)
Interocked - **InterlockedIncrement(), InterlockedDecrement(), InterlockedExchange(), InterlockedCompareExchange(), InterlockedAdd()**
스핀락(Spin Lock)
락프리큐(단일 생상자-단일 소비자용 SPSC, .다중 생산자-다중 소비자용 MPMC)

커널(kernal)

커널은 하드웨어와 소프트웨어 사이에서 중개자 역할로, 오직 커널만이 하드웨어에 직접 접근을 할 수 있다.

.

- 커널의 역할과 특징

    - 독점적 하드웨어 접근
    - 자원 관리자 : CPU, 메모리, 저장장치, 네트워크 등
    - 보안 게이트 키퍼 : 애플리케이션의 모든 시스템 요청을 여기서 검증하고 제어함
    - 서비스 제공자(시스템 콜)

- 커널의 담당
    1. 프로세스 및 스레드 관리
    2. 메모리 관리
    3. 파일 시스템 관리
    4. 네트워크 관리
    5. 디바이스 관리

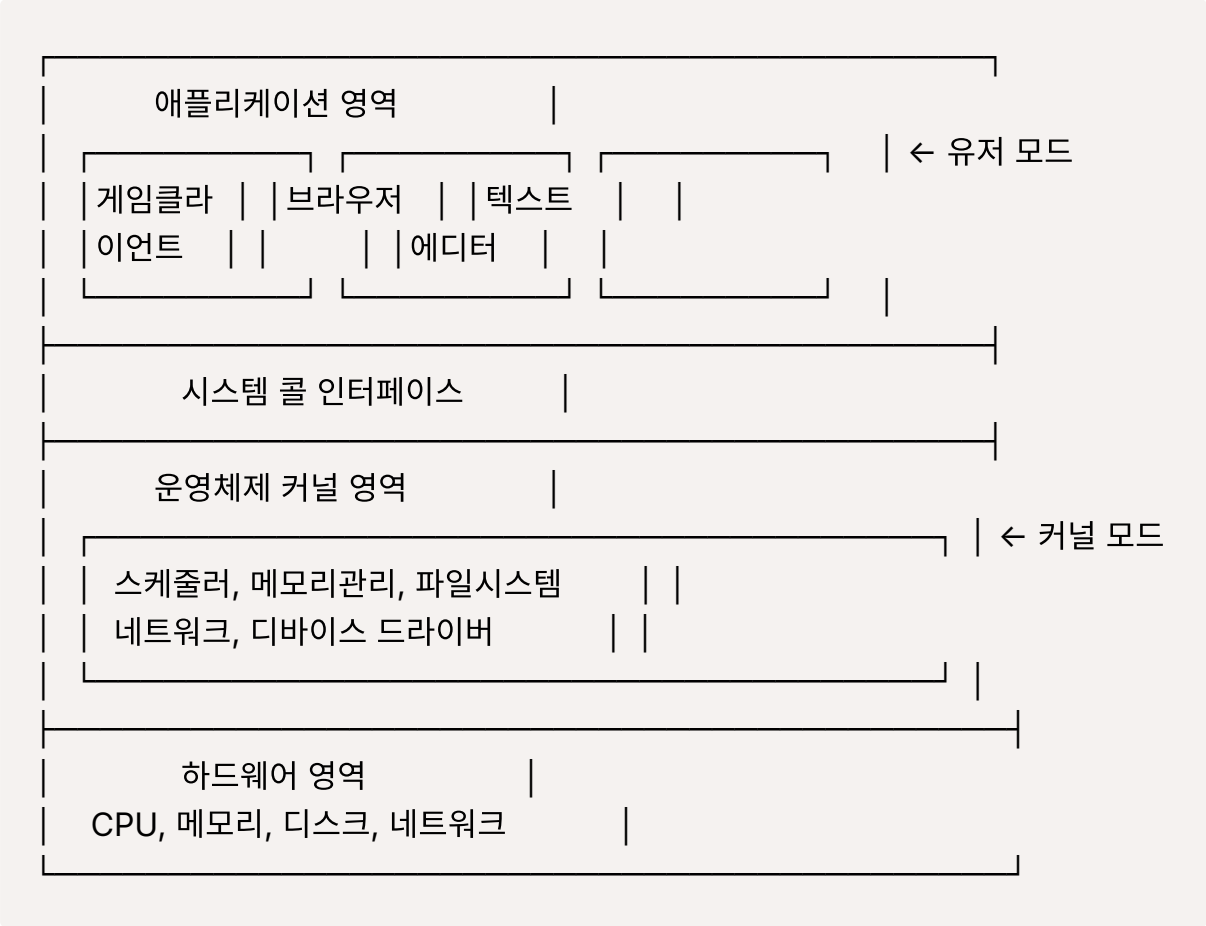
유저모드 vs 커널모드

현대 운영체제는 보안과 안정성을 위해 권한 레벨을 구분해서 동작한다. 우리가 사용하는 소프트웨어가 실행되는 단계인 유저 모드와 운영체제 커널이 실행되는 단계인 커널 모드로 구분된다.

유저모드에서 하드웨어를 사용하기 위해 API 시스템 콜을 통해 내부적으로 커널모드로 전환되고, 커널이 하드웨어에 접근하여 요청을 수행하는 방식이다.

- 유저모드
    - 일반적인 애플리케이션들이 실행되는 모드
    - 하드웨어에 직접 접근이 불가능하고 시스템 콜을 통해 하드웨어를 사용을 요청함
    - 메모리 접근이 제한되며 가상 메모리 영역만 사용함
    - CPU의 특권 명령어를 사용할 수 없음
    - 안전하지만 제한적인 환경
- 커널모드

- 운영체제 커널과 디바이스 드라이버가 실행되는 모드
- 하드웨어에 직접 접근이 가능하며 모든 메모리 영역에 접근 가능
- CPU의 모든 명령어 사용 가능
- 강력하지만 위험한 환경



- 멀티스레드 관점에서 커널은?

  커널은 스레드의 생성부터 종료까지의 관리도 담당한다. 멀티스레드 프로그래밍에서 스레드를 생성하고 관리하는 작업은 대부분 커널의 도움이 필요하다.

```
// C++20 표준 스레드 생성
#include <thread>
#include <iostream>

void worker_function(int id)
 {
```

```
    std::cout << "Worker " << id << " running in user mode\n";
    // 이 함수는 유저 모드에서 실행됨
}

int main() {
    // 스레드 생성 - 내부적으로 커널 모드 전환 발생
    // CreateThread
    std::thread worker1(worker_function, 1);
    std::thread worker2(worker_function, 2);

    // 스레드 종료 대기 - 역시 커널 모드 전환 발생
    // WaitForSingleObject
    worker1.join();
    worker2.join();

    return 0;
}
```

위 코드에서 스레드를 생성하고 종료할때 내부적으로 winAPI함수인 CreateThread, WaitForSingleObject이 호출되는데 이는 시스템콜을 통해 커널 모드로 전환하게 한다.

- 유저모드와 커널모드 전환

  유저모드는 안전하지만 제한적이고, 커널모드는 강력하지만 전환 비용이 있다. 성능 최적화를 위해서는 불필요한 모드 전환을 최소화 하는것이 중요하다!

  - Atomic 연산은 주로 유저모드에서 처리되어 빠르다
  - 뮤텍스 경합은 커널 모드 전환이 필요하다.
  - 시스템콜과 I/O작업은 항상 커널 모드 전환이 필요하다.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <fstream>

std::mutex file_mutex;
```

```
int global_counter = 0;

void worker_function(int id) {
    // A 지점: 변수 할당(유저모드)
    int local_var = id * 2;

    // B 지점: 뮤텍스 락 획득 (커널모드 전환)
    std::lock_guard<std::mutex> lock(file_mutex);

    // C 지점: 전역 변수 수정 (유저모드)
    global_counter++;

    // D 지점: 파일 쓰기 (커널모드 전환)
    std::ofstream file("output.txt", std::ios::app);
    file << "Worker " << id << " executed\n";
}

int main() {
    // E 지점: 스레드 생성(커널모드 전환)
    std::thread t1(worker_function, 1);
    std::thread t2(worker_function, 2);

    // F 지점: 스레드 종료 대기 (커널모드 전환)
    // WaitForSingleObject()
    t1.join();
    t2.join();

    return 0;
}
```

프로세스 vs 스레드

- 프로세스(Process) : 실행중인 프로그램의 인스턴스.
    - 독립된 가상 메모리 공간
- 스레드(Thread) : 프로세스 내에서 실행되는 단위

- 고유한 프로세스 ID(IPD)
- 최소 하나의 스레드(메인 스레드) 포함
- 프로세스 간 직접적인 메모리 공유 불가

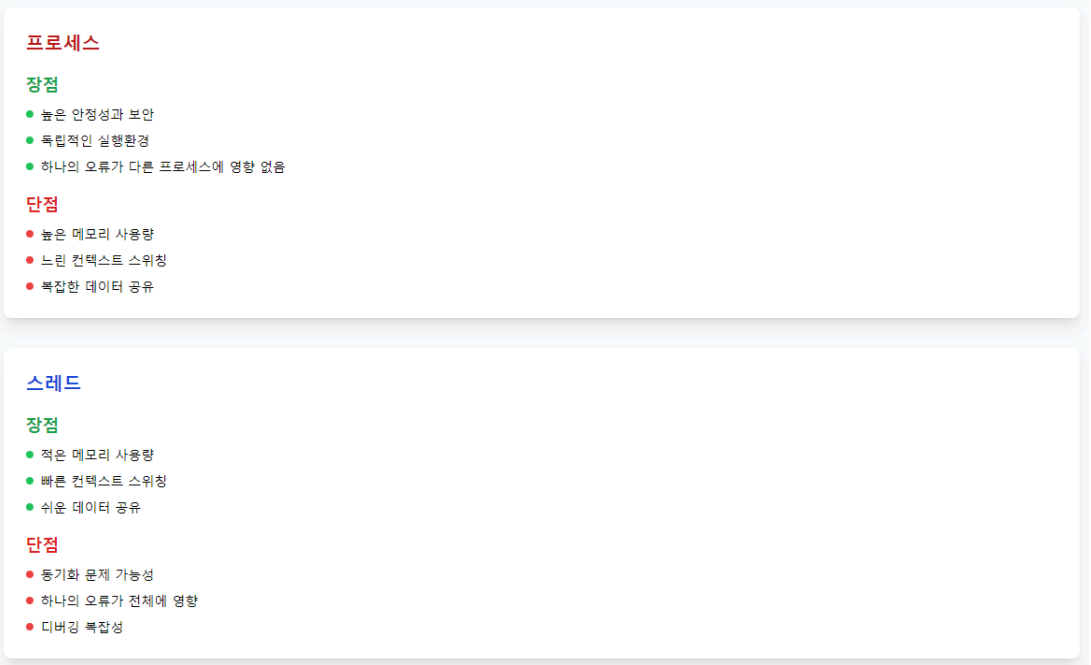
- 같은 프로세스의 스레드들은 프로세스의 메모리 공간 공유
- 고유한 스택과 레지스터 세트
- 빠른 컨텍스트 스위칭
- 스레드 간 데이터 공유 가능

- 프로세스와 스레드 비교
  - 프로세스와 스레드의 차이점

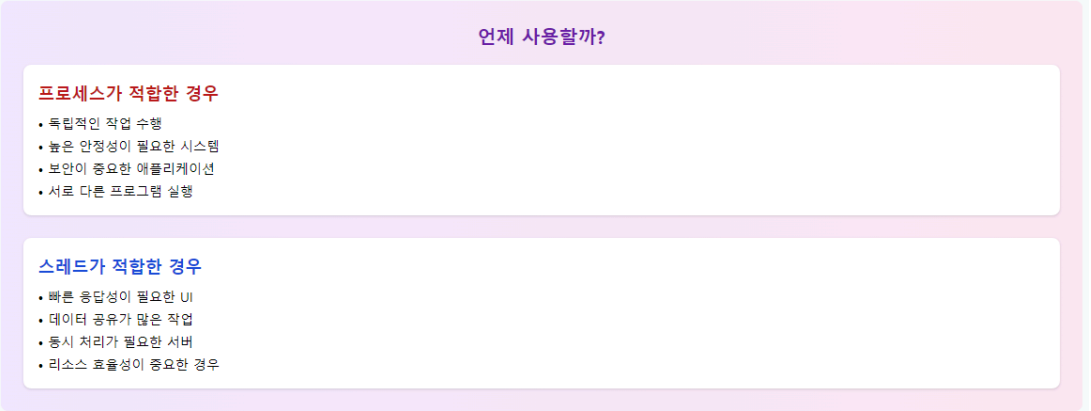
주요 차이점 비교

구분	프로세스	스레드
메모리 공간	독립적인 메모리 공간	프로세스 내 메모리 공간 공유
생성 비용	높음 (메모리 할당 필요)	낮음 (스택만 할당)
컨텍스트 스위칭	느림 (메모리 매핑 변경)	빠름 (레지스터만 교체)
데이터 공유	IPC 필요 (파이프, 소켓 등)	직접 공유 가능
안정성	높음 (독립적)	낮음 (하나 오류시 전체 영향)
동기화	불필요 (독립적)	필요 (공유 자원 접근)

  - 프로세스와 스레드의 장단점



- 멀티스레드 vs 멀티 프로세스



스레드(Thread)

스레드는 프로세스 내에서 실행되는 실행 단위이다. 같은 프로세스의 스레드들은 메모리를 공유하며 각자 독립적인 스택과 레지스터를 지닌다.

- 스레드의 특징
  - 같은 프로세스의 스레드들은 프로세스의 메모리 공간 공유
  - 고유한 스택과 레지스터 세트

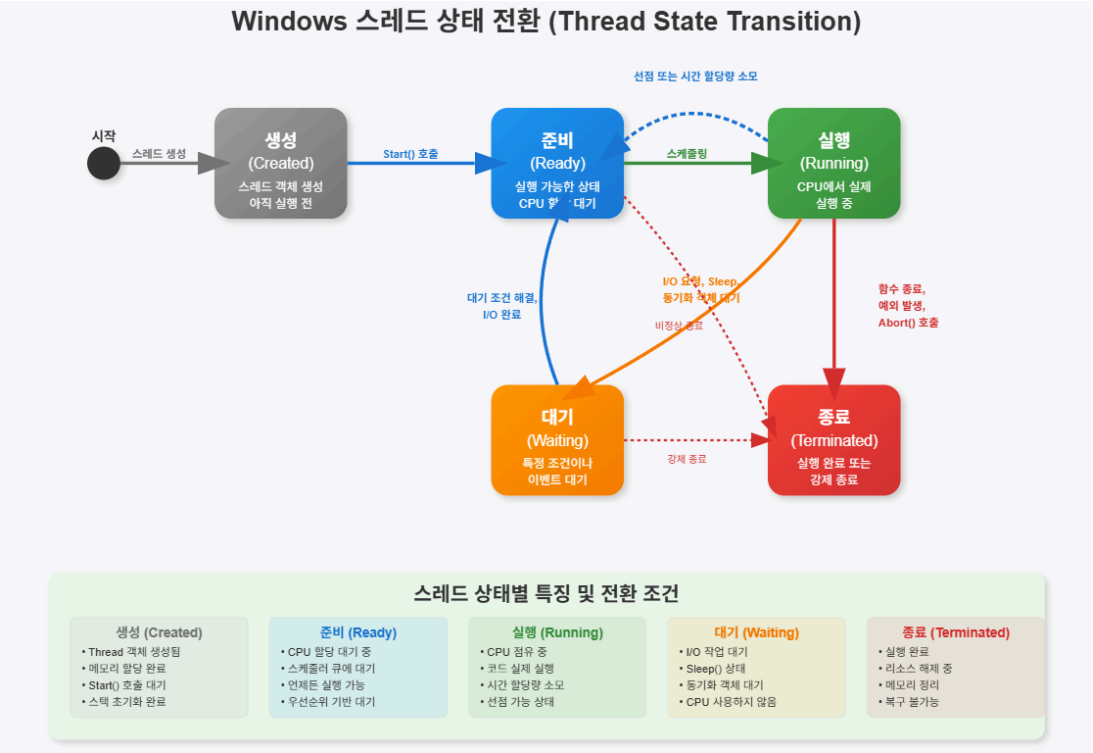
    - 빠른 컨텍스트 스위칭
    - 스레드 간 데이터 공유 가능

- 스레드 구성 요소 (Thread Components)
    - TEB(Thread Environment Block) : 스레드의 모든 메타데이터를 담고있는 핵심 구조체(스레드 ID, 프로세스 ID, 스택 정보, 예외 핸들러 체인)
    - 스택(Stack)
    - 레지스터 컨텍스트 : CPU의 현재 상태를 나타내는 모든 레지스터 값들로 컨텍스트 스위칭시 이 값들이 저장되고 복원된다.
    - 스레드 ID : 시스템 전체에서 해당 스레드를 고유하게 식별하는 번호

- 스레드 스케줄링 특징
    - 시분할 방식(Time Slicing) : 각 스레드에게 정해진 시간 할당량 부여. 시간이 끝나면 다음 스레드로 전환
    - 우선순위 기반 스케줄링 (Priority-Based Scheduling) : windows 스레드 우선순위레벨 0~31을 기반으로 차등 스케줄링하는 시스템
        - 실시간 클래스 (16-31): 시스템 중요 프로세스, 커널 스레드
        - 높은 우선순위 (13-15): 중요한 애플리케이션 스레드
        - 일반 우선순위 (8-12): 대부분의 사용자 애플리케이션
        - 낮은 우선순위 (1-7): 백그라운드 작업
        - 유휴 우선순위 (0): 시스템이 한가할 때만 실행
    - 선점형 스케줄링(Preemptive Scheduling) : 높은 우선순위 스레드가 준비 상태가 되면 현재 실행중인 낮은 우선순위 스레드를 즉시 중단시키고 선점하는 시스템.

- 스레드 라이프 사이클
    1. 생성 (Created): 스레드 객체 생성, 아직 실행 전
    2. 준비 (Ready): 실행 가능한 상태, CPU 할당 대기
    3. 실행 (Running): CPU에서 실제 실행 중
    4. 대기 (Waiting): 특정 조건이나 이벤트 대기

5. 종료 (Terminated): 실행 완료 또는 강제 종료



스레드 생성 함수

1. CreateThread()

windows에서 새로운 스레드를 생성하는 가장 기본적인 API함수로 운영체제 커널에 직접 요청하여 스레드르르 생성하고, 생성된 스레드의 핸들을 반환받는 방식

```
// Create Thread 구조
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  // 보안 특성
    SIZE_T dwStackSize,                        // 스택 크기
    LPTHREAD_START_ROUTINE lpStartAddress,     // 시작 함수
    LPVOID lpParameter,                        // 매개변수
    DWORD dwCreationFlags,                     // 생성 플래그
    LPDWORD lpThreadId                         // 스레드 ID
);

// 스레드 함수
DWORD WINAPI WorkerThread(LPVOID lpParam)
```

```
{
    // 작업 수행
    return 0;  // 정상 종료
}

// Create Thread 사용
HANDLE hThread1 = CreateThread(
    NULL,                   // 기본 보안 특성
    0,                      // 기본 스택 크기 (1MB)
    WorkerThread,           // ⭐ 스레드 함수 DWORD WINAPI WorkerThread(LP
VOID lpParam)
    &threadData1,           // ⭐ 스레드 데이터
    0,                      // 즉시 실행
    &threadId1              // 스레드 ID 받을 변수
);

// Thread 사용 완료 후 반드시 메모리 해제
CloseHandle(hThread1);
```

- CreateThread()의 문제점
  - C 런타임 초기화 누락 : CreateThread로 생성된 스레드는 C 런타임이 초기화 되지 않음

    `CreateThread`는 운영체제 커널 수준에서 직접 스레드를 생성하는 저수준 API이다. 이 방식의 가장 큰 문제는 C 런타임 라이브러리(CRT)가 새로 생성된 스레드에 대해 초기화 작업을 수행하지 않는다는 점이다.
  - 메모리 관리 문제 : malloc/free, new/delete 등 힙 관리 함수들이 제대로 동작하지 않을 수 있다.
  - 전역/정적 변수 초기화 문제 : 스레드별 전역변수나 정적 변수의 초기화가 누락될 수 있다.
  - 예외 처리 문제 : C++예외 처리가 불가능하다.

2. _beginthreadex() ← C++ 권장!

_biginthreadex() 는 C 런타임 라이브러리에서 제공하는 스레드 생성 함수로, C 런타임 초기화를 수행하고 스레드를 생성하는 스레드 생성 함수이다. 내부적으로는 CreateThread함수를 호출한다. 스레드 종료시에도 _endThreadex()를 자동으로 호출하여 정리 작업을 수행한다. (C 런타임을 많이 사용하는 복잡한 애플리케이션에서는 `_endthreadex`를 사용하는 것이 좋다.)

```
// _beginthreadex 구조
uintptr_t _beginthreadex(
    void *security,          // 보안 특성
    unsigned stack_size,     // 스택 크기
    unsigned (__stdcall *start_address)(void *),  // 스레드 함수
    void *arglist,           // 매개변수
    unsigned initflag,       // 생성 플래그
    unsigned *thrdaddr       // 스레드 ID
);

// 스레드 함수
unsigned __stdcall ThreadProc(void* pParam)
{
    // 작업 수행
    _endthreadex(200);  // 종료 코드 200으로 종료. _endthreadex()호출 권장
    return 0;           // 여기는 실행되지 않음. 이것만 해도 정상 종료
}

// _beginthreadex 사용
#include <process.h>

unsigned threadID;
HANDLE hThread = (HANDLE)_beginthreadex(
    NULL,           // 보안 특성
    0,              // 스택 크기
    ThreadProc, // ⭐ 스레드 함수   unsigned __stdcall ThreadProc(void* pPar
am)
    pParam,         // ⭐ 스레드 데이터
    0,              // 생성 플래그
    &threadID       // 스레드 ID
);
```

```
// Thread 사용 완료 후 반드시 메모리 해제
CloseHandle(hThread);
```

- CreateThread() vs _beginthreadex()

구분	CreateThread()	_beginthreadex()
제공자	Windows API	C 런타임 라이브러리
C++ 안전성	제한적	높음
errno 처리	지원 안함	지원
C 런타임 함수	위험할 수 있음	안전함
반환값	HANDLE	uintptr_t (캐스팅 필요)
스레드 함수 시그니처	DWORD WINAPI	unsigned __stdcall

3. std::thread

C++11 표준의 일부로, 모든 초기화와 정리 작업을 자동으로 처리한다. 플랫폼 독립적이며 RAII패턴을 따른다.

```
// 현대적이고 안전한 방법
std::thread myThread(ThreadFunction, param);
myThread.join();
```

주요 스레드 API 함수

스레드 생성 및 관리 API

함수명	용도	사용 빈도
`CreateThread()`	스레드 생성	★★★★★
`ExitThread()`	스레드 종료	★★★★☆

함수명	용도	사용 빈도
`TerminateThread()`	강제 종료 (비권장)	★★☆☆☆
`SuspendThread()`	스레드 일시 중단	★★★☆☆
`ResumeThread()`	중단된 스레드 재시작	★★★☆☆
`GetCurrentThread()`	현재 스레드 핸들	★★★★★
`GetCurrentThreadId()`	현재 스레드 ID	★★★★★

스레드 대기 및 동기화 API

함수명	용도	사용 빈도
`WaitForSingleObject()`	단일 객체 대기	★★★★★
`WaitForMultipleObjects()`	다중 객체 대기	★★★★☆
`Sleep()`	지정 시간 대기	★★★★★
`SwitchToThread()`	다른 스레드에게 양보	★★★☆☆

스레드 우선순위 및 속성 API

함수명	용도	사용 빈도
`SetThreadPriority()`	우선순위 설정	★★★★☆
`GetThreadPriority()`	우선순위 조회	★★★☆☆
`SetThreadAffinityMask()`	프로세서 친화성 설정	★★☆☆☆
`GetExitCodeThread()`	스레드 종료 코드 조회	★★★☆☆

WaitForSingleObject()

```
DWORD WaitForSingleObject
(
  HANDLE hHandle,        // 대기할 객체의 핸들
  DWORD  dwMilliseconds  // 대기 시간 (밀리초)
);
```

- `WAIT_OBJECT_0` : 객체가 신호 상태가 됨
- `WAIT_TIMEOUT` : 시간 초과

- `WAIT_FAILED` : 오류 발생

WaitForMultipleObjects()

```
DWORD WaitForMultipleObjects
(
   DWORD   nCount,        // 핸들 개수
   HANDLE* lpHandles,     // 핸들 배열
   BOOL    bWaitAll,      // TRUE: 모든 객체, FALSE: 하나라도
   DWORD   dwMilliseconds  // 대기 시간
);
```

경쟁조건(Race Condition)

경쟁조건은 여러 스레드가 공유 자원에 동시에 접근할 때, 실행 순서에 따라 결과가 달라지는 상황을 말한다. 이는 매우 위험한 상황으로 아래와 같은 문제를 유발한다.

1. 데이터 불일치(Data Inconsistency)
2. 읽기-수정-쓰기 문제 : 쓰는중에 읽으면 이상한 데이터가 읽어진다. 또 다른 스레드가 쓴 내용을 덮어쓸 수도 있다.

상호 배제(Mutual Exclusion)

상호 배제는 한 번에 하나의 스레드만 공유 자원, 임계 영역(Critical Section)에 접근할 수 있도록 하는 매커니즘이다.

Critical Section이나 Mutex 등으로 구현할 수 있다

원자성(Atomicity)

원자성은 작업이 중간에 중단되지 않고 완전히 실행되거나 전혀 실행되지 않는 특성을 의미한다.

Win32의 `InterlockedIncrement()` 같은 함수들이 원자적 연산을 제공합니다. 원자적 연산은 다른 스레드가 중간 상태를 볼 수 없기 때문에 경쟁조건에서 읽기, 쓰기 연산의 문제를 해결할 수 있다.

```
// 원자성이 보장되지 않는 연산
counter++;  // 실제로는 3단계로 구성됨

// 어셈블리 레벨에서의 실제 동작:
// 1. MOV EAX, [counter]    ; 메모리에서 레지스터로 로드
// 2. INC EAX               ; 레지스터 값 증가
// 3. MOV [counter], EAX    ; 레지스터에서 메모리로 저장
```

데드락 (Deadlock)

데드락은 두 개 이상의 스레드가 서로가 가진 자원을 기다리며 영원히 대기하는 상황이다.

예를들어 스레드 A가 뮤텍스 1을 가지고 뮤텍스 2를 기다리고, 스레드 B가 뮤텍스 2를 가지고 뮤텍스 1을 기다리는 상황을 데드락이라고 한다.

기아상태(Starvation)

기아 상태는 특정 스레드가 계속해서 실행 기회를 얻지 못하는 상황이다.

높은 우선순위의 스레드가 선점당하여 실행되지 못하고 있는 경우가 대표적이다.

가시성(Visbillity)

가시성이란 한 스레드에서 변경한 메모리 값이 다른 스레드에서 언제 보이게 되는가의 문제이다.

CPU 캐시와 컴파일러 최적화 때문에 즉시 보이지 않을 수 있어서 메모리 장벽이나 volatile 키워드가 필요하다.

False Sharing

서로 독립적인 두 변수가 같은 캐시 라인(보통 64바이트)에 위치할 때, 한 스레드가 변수를 수정하면 다른 스레드의 캐시라인까지 무효화되어 성능이 저하되는 현상이다. 구조체 멤버들을 적절히 배치하거나 패딩을 사용해 해결할 수 있다.

동기화 객체

동기화 객체	속도	범위	특징	용도
Critical Section	빠름	프로세스내	가벼움, 재진입 가능	단순한 상호배제
Mutex	보통	프로세스간	무거움, 소유권 개념	프로세스간 동기화
Semaphore	보통	프로세스간	카운터 기반	자원 풀 관리
Event	보통	프로세스간	상태 알림	조건 대기/신호

사용 목적	→ 선택할 동기화 객체
프로세스 내 간단한 상호배제	→ Critical Section
프로세스 간 상호배제	→ Mutex
제한된 자원 풀 관리	→ Semaphore
조건 대기/신호	→ Event

1. Critical Section

임계 영역은 스레드가 동시에 접근할 경우 위험한 경우에 하나의 스레드만 들어올 수 있도록 강제하는 구간이다. Ciritical Section은 Win32에서 제공하는 가장 빠른 동기화 객체로 같은 프로세스 내의 스레드들 간의 동기화에만 사용할 수 있다.

- `InitializeCriticalSection()` : 임계 영역 초기화
- `EnterCriticalSection()` : 임계 영역 진입
- `LeaveCriticalSection()` : 임계 영역 해제
- `DeleteCriticalSection()` : 임계 영역 삭제

```
#include <windows.h>
#include <iostream>
#include <thread>

class ThreadSafeCounter {
private:
    CRITICAL_SECTION cs;
    int count;

public:
    ThreadSafeCounter() : count(0)
    {
        // InitializeCriticalSection() : 임계 영역 초기화
        InitializeCriticalSection(&cs);
    }

    ~ThreadSafeCounter()
    {
        // DeleteCriticalSection() : 임계 영역 삭제
        DeleteCriticalSection(&cs);
    }

    void Increment() {
        // EnterCriticalSection() : 임계 영역 진입 ~ 여기부터는 하나의 스레드만 실행 가능
        EnterCriticalSection(&cs);

        int temp = count;
        Sleep(1);               // 경쟁상황 시뮬레이션. 대기상태에 들어가더라고 이 구간에 다른 스레드가 들어올 수 없다.
        count = temp + 1;
```

```
        // LeaveCriticalSection() : ~ 임계 영역 해제
        LeaveCriticalSection(&cs);
    }

    int GetCount() const {
        return count;
    }
};
```

2. Mutex

Mutex(Mutual Exclusion)는 상호 배제 객체로 프로세스 간 동기화도 가능한 커널 객체이다. Ciritical Section보다 무겁지만 더 강력한 기능을 제공한다.

- `CreateMutex()` : 뮤텍스 생성
- `WaitForSingleObject()` : 뮤텍스 획득 대기 (임계 영역에 들어갈 권한)
- `ReleaseMutex()` : 뮤텍스 반납
- `CloseHandle()` : 뮤텍스 핸들 해제

```
#include <windows.h>
#include <iostream>
#include <vector>
#include <thread>

class MutexExample {
private:
    HANDLE hMutex;
    int sharedResource;

public:
    MutexExample() : sharedResource(0) {
        // CreateMutex() : 뮤텍스 생성
        hMutex = CreateMutex(
            NULL,    // 보안 속성
            FALSE,   // 초기 소유권 없음
```

```
            NULL     // 이름 없음 (프로세스 내부용)
        );

        if (hMutex == NULL) {
            throw std::runtime_error("뮤텍스 생성 실패");
        }
    }

    ~MutexExample() {
        if (hMutex) {
                // CloseHandle() : 뮤텍스 핸들 해제
            CloseHandle(hMutex);
        }
    }

    void AccessResource(int threadId) {
        // WaitForSingleObject() : 뮤텍스 획득 대기
        // OS가 지금 뮤텍스를 누가 사용중인지 체크하여 아무도 사용중이 아니라면 이
스레드에게 권한을 줌
        DWORD waitResult = WaitForSingleObject(hMutex, 5000);

        switch (waitResult) {
        case WAIT_OBJECT_0:
            // 뮤텍스 획득 성공
            // 임계영역 시작 ~
            std::cout << "스레드 " << threadId << ": 리소스 접근 시작\n";

            int oldValue = ㅊ;
            Sleep(1000);
            sharedResource = oldValue + 1;

            std::cout << "스레드 " << threadId << ": 리소스 값 = "
                    << sharedResource << "\n";

            // ReleaseMutex() : 뮤텍스 해제
            // ~ 임계 영역 종료
            if (!ReleaseMutex(hMutex)) {
                std::cerr << "뮤텍스 해제 실패\n";
```

```
            }
            break;

        case WAIT_TIMEOUT:
            std::cout << "스레드 " << threadId << ": 타임아웃 발생\n";
            break;

        case WAIT_FAILED:
            std::cerr << "스레드 " << threadId << ": 대기 실패\n";
            break;
        }
    }

    int GetResource() const { return sharedResource; }
};
```

3. Semaphore

세마포어는 제한된 수의 자원에 대한 접근을 제어하는 동기화 객체이다. Ciritical Section이나 Mutex는 하나의 스레드만 임계 영역에 접근이 가능하게 제어했지만, Semaphore는 여러 스레드가 동시에 접근할 수 있다.

- `CreateSemaphore()` : 세마포어 생성
- `WaitForSingleObject()` : 세마포어 획득 대기 (임계 영역에 들어갈 권한)
- `ReleaseSemaphore()` : 세마포어 반납
- `CloseHandle()` : 세마포어 핸들 해제

```
#include <windows.h>
#include <iostream>
#include <vector>
#include <process.h> // _beginthreadex, _endthreadex를 위해 필요
```

```
class ConnectionPool
{
private:
    HANDLE hSemaphore;
    static const int MAX_CONNECTIONS = 3;

public:
    ConnectionPool() {
        // CreateSemaphore() : 세마포어 생성
        hSemaphore = CreateSemaphore(
            NULL,           // 보안 속성
            MAX_CONNECTIONS,   // 초기 카운트 : 처음에 수용 가능한 스레드 수
            MAX_CONNECTIONS,   // 최대 카운트 : 최대로 수용 가능한 스레드 수
            NULL            // 이름
        );

        if (hSemaphore == NULL) {
            throw std::runtime_error("세마포어 생성 실패");
        }
    }

    ~ConnectionPool() {
        if (hSemaphore) {
                // CloseHandle() : 세마포어 핸들 해제
            CloseHandle(hSemaphore);
        }
    }

    void UseConnection(int threadId) {
        std::cout << "스레드 " << threadId << ": 연결 요청\n";

        // WaitForSingleObject() : 세마포어 획득 대기
        DWORD waitResult = WaitForSingleObject(hSemaphore, INFINITE);

        if (waitResult == WAIT_OBJECT_0) {
                // 세마포어 획득
                // ~ 임계 영역 진입
```

```
            std::cout << "스레드 " << threadId << ": 연결 획득\n";
            Sleep(2000 + (rand() % 3000));
            std::cout << "스레드 " << threadId << ": 연결 해제\n";

            // ReleaseSemaphore() : 세마포어 해제 (연결 반납)
            ReleaseSemaphore(hSemaphore, 1, NULL);
        }
    }
};

// 스레드 구조체
struct ThreadData {
    ConnectionPool* pool;
    int threadId;
};

// 스레드 함수
unsigned int __stdcall ThreadFunction(void* pArguments) {
    ThreadData* data = static_cast<ThreadData*>(pArguments);

    if (data) {
        data->pool->UseConnection(data->threadId);
    }
    delete data; // 동적으로 할당된 데이터 해제
    _endthreadex(0);
    return 0;
}

// 사용 예제
void TestConnectionPool()
{
    ConnectionPool pool;
    const int NUM_THREADS = 5;
    std::vector<HANDLE> threadHandles;

    // 5개 스레드가 3개 연결을 두고 경쟁
    for (int i = 1; i <= NUM_THREADS; ++i) {
        // 스레드에 전달할 데이터를 동적으로 생성
```

```
        ThreadData* data = new ThreadData{&pool, i};

        HANDLE hThread = (HANDLE)_beginthreadex(
            NULL,                   // 보안 속성
            0,                  // 스택 크기 (0 = 기본값)
            ThreadFunction,         // 스레드 함수
            data,                 // 스레드 함수에 전달할 인자
            0,                  // 생성 플래그 (0 = 즉시 실행)
            NULL                    // 스레드 ID (필요 없는 경우 NULL)
        );

        if (hThread)  threadHandles.push_back(hThread);
        else          delete data; // 스레드 생성 실패 시 메모리 해제

    }

    // 모든 스레드가 끝날 때까지 대기
    WaitForMultipleObjects(threadHandles.size(), threadHandles.data(), TRU
E, INFINITE);

    // 스레드 핸들 닫기
    for (HANDLE h : threadHandles) {
        CloseHandle(h);
    }
}

int main()
{
    srand(time(NULL));
    TestConnectionPool();
    return 0;
}
```

4. Event (Manual/Auto Reset)

Event 객체는 특정 조건이나 상태 변화를 알리는데 사용되는 동기화 객체이다. Event 객체는 Auto_Reset Event와 Manual-Reset Event가 있으며 signaled 상태가 되는 조건에 차이가 있다.

Critical Section, Mutex, Shmephore는 임계 영역 보호 목적으로 사용되는 동기화 객체라면 Event는 단순 스레드의 시작/대기/신호 용도로 사용하는 동기화 객체이다.

```
// 이벤트 객체 생성 함수
HANDLE CreateEvent
(
  NULL,          // 보안 속성
    TRUE,          // TRUE : 수동 리셋, FALSE : 자동 리셋
  TRUE,          // 초기 신호 상태. TRUE : signaled , FALSE : non-signaled
  NULL,          // 이름
);
```

- signaled 상태 : 초록불. 대기하던 스레드들이 깨어나서 다음 작업을 진행할 수 있는 상태
- non-signaled 상태 : 빨간불. 스레드들이 대기해야하는 상태

  → 스레드들이 Event 객체를 기다리고 있을 때, Event가 Signaled 상태가 되면 "이제 작업을 계속 진행해도 된다"는 신호를 받는 것이다.

  Non-Signaled 상태에서는 스레드들이 계속 기다리고, Signaled 상태가 되면 기다리던 스레드들이 작업을 재개할 수 있게 된다.

1. Auto_Reset Event (자동 리셋 이벤트)

   하나의 대기 스레드가 깨어나면 자동으로 non-signaled 상태로 변경

   문을 한 번만 열어주는 자동문 같은 개념이다. 한 사람이 문을 통과하면 문이 자동으로 닫혀버린다. 즉, 대기하고 있던 스레드 중 하나가 신호를 받아 깨어나면, 그 즉시 다시 잠긴 상태가 되어서 다른 스레드들은 계속 기다려야 한다.

   - `CreateEvent()` : 이벤트 생성
   - `SetEvent()` : signaled 신호 전달 → 하나의 스레드만 깨어나고 자동으로 non-signaled 상태가 됨
   - `WaitForSingleObject()` : 이벤트 객체가 Signaled상태가 되는 것을 대기

- `CloseHandle()` : 이벤트 핸들 해제

```
#include <windows.h>
#include <iostream>
#include <vector>
#include <stdexcept>
#include <process.h>

class WorkManager
{
private:
    HANDLE hWorkEvent;
    volatile bool shouldStop;

public:
    WorkManager() : shouldStop(false)
    {
        // CreateEvent() : 자동 리셋 이벤트 생성 (FALSE, FALSE : 자동리셋, 초기상태 non-signal)
        hWorkEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

        if (!hWorkEvent) throw std::runtime_error("이벤트 생성 실패");
    }

    ~WorkManager()
    {
        // CloseHandle() : 이벤트 핸들 정리
        if (hWorkEvent) CloseHandle(hWorkEvent);
    }

    void WorkerThread(int workerId)
    {
        while (!shouldStop)
        {
            std::cout << "워커 " << workerId << ": 작업 신호 대기 중...\n";

                    // WaitForSingleObject() : Signaled 상태가 되면 하나의 스레드가 깨어나고 곧바로 Non-Signaled상태로 변경
```

```
            WaitForSingleObject(hWorkEvent, INFINITE);

            if (shouldStop) break;

            std::cout << "워커 " << workerId << ": 작업을 할당받아 수행합니
다.\n";
            Sleep(1500);
            std::cout << "워커 " << workerId << ": 작업 완료.\n";
        }
        std::cout << "워커 " << workerId << ": 종료.\n";
    }
    void SignalWork()
    {
        // SetEvent() : 이벤트를 Signaled 상태로 변경 -> 대기 중인 스레드 중 하
나만 깨어남
        SetEvent(hWorkEvent);
    }

    void StopAllWorkers(int numWorkers)
    {
        shouldStop = true;
        // 대기 중인 모든 스레드를 깨워 루프를 종료시키기 위해 여러 번 호출
        for (int i = 0; i < numWorkers; ++i) {
            SetEvent(hWorkEvent);
        }
    }
};

// 스레드 데이터
struct ThreadData
{
    WorkManager* manager;
    int workerId;
};

// 스레드 함수
unsigned int __stdcall ThreadFunction(void* pArguments)
{
```

```
    ThreadData* data = static_cast<ThreadData*>(pArguments);
    if (data) {
        // 실제 클래스 멤버 함수 호출
        data→manager→WorkerThread(data→workerId);
        delete data; // 동적으로 할당된 데이터 해제
    }
    _endthreadex(0); // 스레드 종료
    return 0;
}

int main()
{
    const int NUM_WORKERS = 3;
    const int NUM_TASKS = 5;
    WorkManager manager;
    std::vector<HANDLE> workerHandles;

    // 워커 스레드 생성 및 시작
    for (int i = 0; i < NUM_WORKERS; ++i)
    {
        ThreadData* data = new ThreadData{ &manager, i + 1 };

        HANDLE hThread = (HANDLE)_beginthreadex(
            NULL,              // 보안 속성
            0,                 // 스택 크기 (0 = 기본값)
            ThreadFunction,    // 스레드 함수
            data,              // 스레드 함수에 전달할 인자
            0,                 // 생성 플래그 (0 = 즉시 실행)
            NULL               // 스레드 ID 포인터
        );

        if (hThread)
        {
            workerHandles.push_back(hThread);
        }
        else
        {
            delete data;
```

```
            std::cerr << "스레드 " << i + 1 << " 생성 실패!" << std::endl;
        }
    }

    Sleep(100);

    // 5개의 작업을 순차적으로 요청
    for (int i = 0; i < NUM_TASKS; ++i)
    {
        std::cout << "\n[메인] " << i + 1 << "번째 작업 신호를 보냅니다. (SetEv
ent)\n";
        manager.SignalWork();
        Sleep(500);
    }

    std::cout << "\n[메인] 모든 워커 종료 신호를 보냅니다.\n";
    manager.StopAllWorkers(NUM_WORKERS);

    WaitForMultipleObjects(workerHandles.size(), workerHandles.data(),
TRUE, INFINITE);
    std::cout << "\n[메인] 프로그램이 종료되었습니다.\n";



    // 스레드 핸들 정리
    for (HANDLE hThread : workerHandles)
    {
        CloseHandle(hThread);
    }

    return 0;
}
```

2. Manual-Reset Event (수동 리셋 이벤트)

   명시적으로 ResetEvent()를 호출할 때까지 signaled 상태 유지

   수동으로 열고 닫는 일반 문 같은 개념이다. 문을 열어두면 여러 사람이 계속 통과할 수 있고, 누군가가 직접 문을 닫을 때까지는 계속 열린 상태를 유지한다. 즉, 신호가 발생하

면 대기 중인 모든 스레드가 깨어날 수 있고, 개발자가 명시적으로 ResetEvent()를 호출해서 다시 잠글 때까지는 계속 열린 상태이다.

- `CreateEvent()` : 이벤트 생성
- `SetEvent()` : signaled 신호 전달 → 모든 스레드들이 깨어남
- `ResetEvet()` : signaled 신호 해제 → non-signaled 상태가 됨
- `WaitForSingleObject()` : 이벤트 객체가 Signaled상태가 되는 것을 대기
- `CloseHandle()` : 이벤트 핸들 해제

```
#include <windows.h>
#include <iostream>
#include <vector>
#include <stdexcept>
#include <process.h>


class WorkerController
{
private:
    HANDLE hStartEvent;
    volatile bool shouldStop;

public:
    WorkerController() : shouldStop(false)
    {
        // CreateEvent() : 수동 리셋 이벤트 생성 (TRUE, FALSE : 수동 리셋, 초기 상태 Non-Signaled)
        hStartEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
        if (!hStartEvent) {
            throw std::runtime_error("이벤트 생성 실패");
        }
    }

    ~WorkerController()
    {
        // CloseHandle() : 이벤트 핸들 해제
        if (hStartEvent) CloseHandle(hStartEvent);
```

```
    }

    void WorkerThread(int workerId)
    {
        std::cout << "워커 " << workerId << ": 시작 신호 대기 중...\n";

        // WaitForSingleObject() : hStartEvent가 Signaled 상태가 될 때까지 무
한 대기
        WaitForSingleObject(hStartEvent, INFINITE);

        // signaled 상태 진입. 작업 시작
        if (shouldStop)
        {
            std::cout << "워커 " << workerId << ": 중지 신호를 받아 종료합니
다.\n";
            return;
        }

        std::cout << "워커 " << workerId << ": 작업을 시작합니다.\n";
        Sleep(1000 + (rand() % 1000));
        std::cout << "워커 " << workerId << ": 작업 완료.\n";
    }

    void StartAllWorkers()
    {
        std::cout << "\n[메인] 모든 워커에게 시작 신호를 보냅니다. (SetEvent)
\n";
        // SetEvent() : 이벤트를 Signaled 상태로 변경 -> 대기 중인 모든 스레드
가 동시에 깨어남
        SetEvent(hStartEvent);
    }

    void StopAllWorkers()
    {
        shouldStop = true;
        // SetEvent() : 이벤트를 Signaled 상태로 변경 -> 대기 중인 모든 스레드
가 동시에 깨어남
        SetEvent(hStartEvent);
```

```
    }
};

// 스레드 데이터
struct ThreadArgs
{
    WorkerController* controller;
    int workerId;
};

// 스레드 함수
unsigned int __stdcall WorkerThreadFunc(void* pArgs)
{
    ThreadArgs* args = static_cast<ThreadArgs*>(pArgs);
    if (args)
    {
        args->controller->WorkerThread(args->workerId);
        delete args;
    }
    _endthreadex(0);
    return 0;
}


int main()
{
    const int NUM_WORKERS = 5;
    WorkerController controller;
    std::vector<HANDLE> workerHandles;
    workerHandles.reserve(NUM_WORKERS);

    // 스레드 및 워커 생성
    for (int i = 0; i < NUM_WORKERS; ++i) {
        ThreadArgs* args = new ThreadArgs{&controller, i + 1};

        HANDLE hThread = (HANDLE)_beginthreadex(
            NULL,              // 보안 속성
            0,                 // 스택 크기
```

```
            WorkerThreadFunc,   // 스레드 함수
            args,               // 스레드 함수에 전달할 인자
            0,                  // 생성 플래그
            NULL                // 스레드 ID 변수
        );

        if (hThread)
        {
            workerHandles.push_back(hThread);
        }
        else
        {
            delete args;
            std::cerr << "스레드 " << i + 1 << " 생성 실패\n";
        }
    }

    Sleep(1000);

    // signaled 상태 on, 워커들 작업 시작
    controller.StartAllWorkers();

    if (!workerHandles.empty())
    {
        WaitForMultipleObjects(workerHandles.size(), workerHandles.data
(), TRUE, INFINITE);
    }
    std::cout << "\n[메인] 모든 작업이 완료되었습니다.\n";

    // 스레드 핸들 정리
    for (HANDLE h : workerHandles)
    {
        CloseHandle(h);
    }

    return 0;
}
```

흐름제어 (Flow Control)

흐름제어는 여러 스레드가 동시에 실행될 때 작업들 간의 실행 순서, 속도, 동기화 방식을 조절하는 개념이다. 이벤트나 세마포어 등으로 구현할 수 있다.

- 멀티스레드 환경에서 발생할 수 있는 문제들
  - 경쟁 조건 : 여러 스레드가 동시에 공유 자원에 접근할 때 잘못된 값이 발생하는 문제
  - 교착 상태 : 두개 이상의 스레드가 서로 가진 자원을 기다리며 무한 대기하는 상태
  - 기아 상태 : 우선순위나 스케줄링때문에 실행이 되지 못하고있는 상태
  - 속도 불균형 : 어떤 스레드는 너무 빨리, 어떤 스레드는 너무 느리게 실행되어 시스템 병목이 발생

- 흐름 주요의 주요 매커니즘
  - 동기화 - Critical Section, Mutex, Semaphore, Event
  - 신호 - Event, Condition Variable
  - 스케줄링 - OS스케줄러, Sleep, 우선순위 지정
  - 속도 조절 - 큐 버퍼 크기 제한, Semaphore
  - 파이프라인 제어 : 큐, Event, Semaphore

- 흐름제어가 필요한 대표적인 상황들
  - 생산자 - 소비자 패턴 : 데이터를 생성하는 생산자 스레드와 데이터를 소비하는 소비자 스레드간의 속도 조절
  - 리더 - 팔로워 패턴 : 여러 스레드 중 하나만 리더가 되어 작업을 수행
  - 파이프라인 처리 : 여러 단계의 작업을 순차적으로 처리

생산자-소비자 패턴 (Projucer-Consumer)

생산자 소비자 패턴은 데이터를 생성하는 스레드와 데이터를 처리하는 스레드가 서로 협력하는 구조다. 이때 생산 속도와 소비 속도가 다를 수 있기 때문에 버퍼(큐)를 두어 균형을 맞춘다.

- 동기화 도구
  - Critical Section : 큐 접근 보호
  - Event : 대기, 신호 처리

```
// 생산자
unsigned __stdcall Producer(void*) {
    for (int i = 0; i < 5; i++) {
        Sleep(500);

        // 데이터 생산
        EnterCriticalSection(&cs);
        buffer.push(i);
        std::cout << "Produced: " << i << std::endl;
        LeaveCriticalSection(&cs);

        // 소비자 깨우기 (신호 상태)
        SetEvent(hEvent);
    }

    return 0;
}

// 소비자
unsigned __stdcall Consumer(void*) {
    while (true) {
        // Producer가 아이템을 하나 이상 넣었다는 신호를 기다림 (자동 리셋 이벤트)
        WaitForSingleObject(hEvent, INFINITE);

        //  데이터 소비
        EnterCriticalSection(&cs);
        while (!buffer.empty())
        {
```

```
            int data = buffer.front();
            buffer.pop();
            std::cout << "Consumed: " << data << std::endl;
        }
        LeaveCriticalSection(&cs);
    }
    return 0;
}
```

리더-팔로워 패턴 (Projucer-Consumer)

리더 팔로워 패턴은 여러 스레드 중 하나만 리더가 되어 작업을 수행한다. 이때 핵심 아이디어는 여러 스레드가 이벤트를 기다리면서 발생하는 비효율(thundering Herd Problem)을 막는 것이다. 오직 하나의 '리더' 스레드만이 이벤트를 기다리고 나머지 '팔로워' 스레드들은 리더가 될 차례를 기다린다.

```
flowchart LR
    E[이벤트 소스] --> L[리더 스레드]
    L -->|작업 처리| Done[완료]
    L -->|리더 역할 해제| F1[팔로워 스레드]
    F1 -->|리더 승격| L
```

- 자동 리셋 이벤트(Event)로 리더 자격 얻고, 리더 자격 얻은애가 세마포어(Semaphore) 획득 대기로 작업 대기. 세마포어 획득시(처리할 작업이 생기면) SetEvent를 통해 다른 팔로워들에게 리더자격을 넘기고 본인은 이제 리더가 아니므로 할당된 작업을 처리(Critical Section)

파이프라인 처리

데이터를 여러 단계로 나누어 각 단께마다 다른 스레드가 처리. 병렬 처리로 전체 처리량이 증가한다는 장점이 있음

각 단계마다 이벤트를 만들어서 각자 앞 단계의 이벤트를 기다리는 방식

```
flowchart LR
    S1[단계1: 데이터 로드] --> S2[단계2: 디코딩]
    S2 --> S3[단계3: 렌더링]
```

Interlocked

Interlocked 함수는 원자적(automic)연산을 제공하여 락 없이도 스레드 안전한 연산을 수행할 수 있게 한다. 이는 성능상 매우 유리하며 간단한 동기화에 적합하다. 함수 이름에 Exchange가 들어가면 이전의 값을 반환하고, 이외의 함수는 연산 후의 값을 반환한다.

```
// 1. InterlockedIncrement / InterlockedDecrement
// 공유 변수 값을 원자적으로 1증가/감소한다. 반환값 : 연산 후의 값
LONG InterlockedIncrement(LONG volatile* Addend);
LONG InterlockedDecrement(LONG volatile* Addend);

// 2. InterlockedExchange
// Tartget을 value로 원자적으로 교체한다. 반환값 : 교체 이전의 값
LONG InterlockedExchange(LONG volatile* Target, LONG Value);

// 3. InterlockedCompareExchange (CAS - Compare And Swap)
// Destination이 Comparand와 같으면 Exchange로 원자적으로 교체한다. 반환값 : 교체 이전의 값
LONG InterlockedCompareExchange(
    LONG volatile* Destination,
    LONG Exchange,
    LONG Comparand
);

// 4. InterlockedAdd
// Addend에 value를 원자적으로 더한다. 반환값 : 연산 후의 값
LONG InterlockedAdd(LONG volatile* Addend, LONG Value);

// 5. InterlockedExchangePointer
```

```
// Target포인터를 Value(포인터)로 원자적으로 교체한다. 반환값 : 교체 이전의 포인터
PVOID InterlockedExchangePointer(
    PVOID volatile* Target,
    PVOID Value
);

// 6. InterlockedCompareExchangePointer (포인터 CAS - Compare And Swap)
// Destination이 COmparand와 같다면 Exchange로 교체. 반환값 : 교체 이전의 포인터
PVOID InterlockedCompareExchangePointer(
    PVOID volatile* Destination,
    PVOID Exchange,
    PVOID Comparand
);

// 7. InterlockedExchangeAdd
// 교체 후에 Addend에 value를 원자적으로 더한다. 반환값 : 연산 전의 값
LONG InterlockedExchangeAdd(LONG volatile* Addend, LONG Value);


// MemoryBarrier
// CPU 명령 재정렬을 방지하는 풀 펜스
// 다중 CPU 코어 환경에서 lock-free 알고리즘의 일관성을 보장한다.
VOID MemoryBarrier();


// 비트 연산
LONG InterlockedAnd(LONG volatile* Destination, LONG Value);
LONG InterlockedOr(LONG volatile* Destination, LONG Value);
LONG InterlockedXor(LONG volatile* Destination, LONG Value);


// 64비트 버전
LONGLONG InterlockedIncrement64(LONGLONG volatile* Addend);
LONGLONG InterlockedDecrement64(LONGLONG volatile* Addend);
LONGLONG InterlockedExchange64(LONGLONG volatile* Target, LONGLO
```

```
NG Value);
LONGLONG InterlockedCompareExchange64(
    LONGLONG volatile* Destination,
    LONGLONG Exchange,
    LONGLONG Comparand
);
LONGLONG InterlockedAdd64(LONGLONG volatile* Addend, LONGLONG V
alue);
```

Volatile

volatile은 타입 한정자(cv-qualifier)로서 "이 객체에 대한 읽기/쓰기 접근 자체가 외부 세계에 의미가 있으니 컴파일러가 마음대로 제거하거나 합치거나 캐시하지말라"라는 신호를 준다. 예를 들어 컴파일러가 최적화를 위해 const변수를 상수로 캐시에 올려 상수값을 사용하는데, 외부에서 그 const변수의 const를 없애서 변수에 다른 값을 넣는다면 연산이 원하는대로 이루어지지 않는다. 이때 volatile 키워드를 사용하여 컴파일러가 마음대로 최적화를 하지 말라고 선언하는 것이다.

- **하드웨어 레지스터, 폴링 루프, 특정 저수준 상호작용**에만 사용해야 한다.
- **스레드 동기화에는 절대 사용하지 말고** `std::atomic`을 사용해야 한다.
- Win32의 `volatile`은 언어적으로 C++의 `volatile`과 같지만, **Windows의 메모리 모델 및 API 컨벤션 안에서 "공유 메모리 접근용"으로 확장된 의미로 사용**된다.
- 실제 동기화나 원자성은 `Interlocked` 함수 또는 명시적 락이 담당한다.
- 즉, **Win32의 `volatile`은 "최적화를 막는 보조 표시" + "공유 변수임을 나타내는 코드 컨벤션"**이다.

스핀락

Lock-Free 큐