작성자(학번 이름) : 2019180052 한창우

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **주차** | 9주차 | **기간** | 2024.8. 19~ 8. 25 | **지도교수** |  |
| 이번주 한일 요약 | Windows Via C/C++  챕터 10 “동기 및 비동기 장치 I/O” | | | | |

<상세 수행내용>

**챕터 10: 동기 및 비동기 장치 I/O**

이번 챕터에서는 고성능, 확장성, 응답성, 그리고 안정성 등을 고려하여 애플리케이션을 개발할 수 있는 마이크로소프트 윈도우의 기술들에 대해 알아본다.

**확작성**이 좋은 애플리케이션이란 많은 수의 동시 작업을 처리할 수 있는 애플리케이션을 말한다.

서비스 애플리케이션에서 **동시 작업**이란 예측 불가능한 시점에 들어오은 클라이언트의 요청을 처리하는 것을 의미한다. 그리고 이런 요청은 네트워크 어댑터와 같은 I/O 장치를 통해 전달된다.

마이크로소프트 윈도우는 스레드를 이용하여 작업을 세분화할 수 있는 훌륭한 기능을 제공하고 있다. 스레드는 단일 프로세서에 의해 구동되기 때문에 다수의 프로세서를 가진 환경에서는 여러 개의 스레드가 동시에 수행될 수 있어서 동시 작업을 처리하기 효율적이다. 그러나 스레드가 동기적인 I/O작업을 요청하면 스레드는 블로킹 상태가 되고, 이는 수행 성능에 나쁜 영향을 미치게 된다. 따라서 가능하면 스레드가 블로킹되지 않는 것이 좋다. 스레드들이 계속해서 수행되려면 스레드들 간에 수행할 작업에 대해 상호 통신해야 할 필요가 있다.

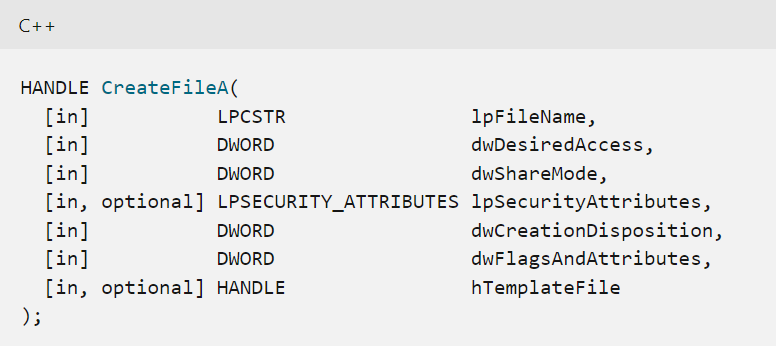
그래서 마이크로소프트에서 매우 잘 정제된 스레드 간의 통신 메커니즘을 개발하였다. 이 메커니즘이 바로 고성능의 확장성 있는 애플리케이션을 개발할 수 있게 해주는 **“I/O 컴플리션 포트 (completion port)”** 이다. I/O 컴플리션 포트를 이용하면 장치에 대한 읽기와 쓰기를 수행할 때 응답을 대기할 필요가 없으므로 애플리케이션의 성능이 놀랄 만큼 개선된다.

이 책에서는 I/O 컴플리션 포트의 활용 예가 장치 I/O에 구한되었지만, I/O 컴플리션 포트는 다방면에서 활용될 수 있는 훌륭한 스레드간 통신 메커니즘이라고 소개하고 있다.

**장치 열기와 닫기**

윈도우 운영체제의 강점 중 하나는 다양한 종류의 장치들을 사용할 수 있다는 것이다.

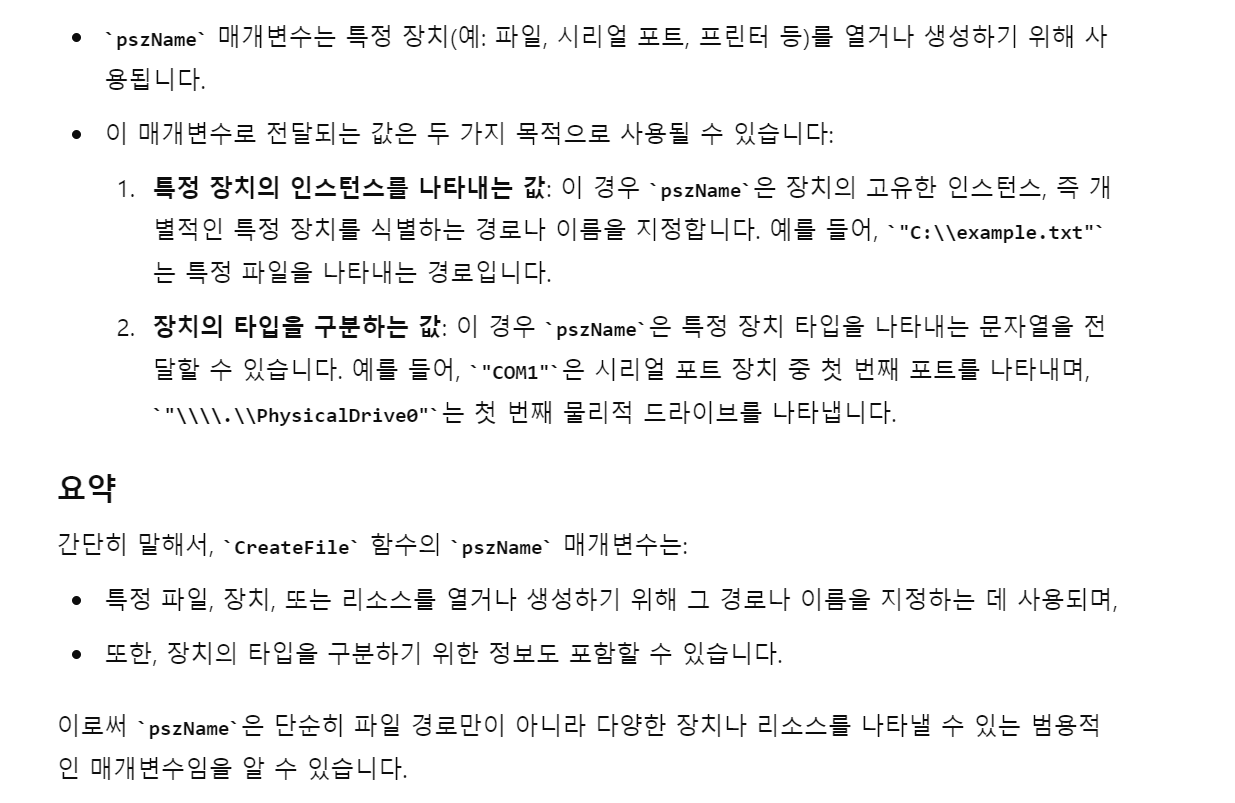
(**장치**: 총신 가능 어떤 것)



CreateFile 함수를 이용하면 디스크에 새로운 파일을 생성하거나 기존 파일에 대한 열기를 수행할 수 있을 뿐만 아니라 파일이 아닌 다른 장치에 대해서도 열기 작업을 수행할 수 있다.

각 매개변수에 대하여 값을 전달하면 어떤 제어를 할 수 있는지 간략하게 요약해봤다.

**pszName**: 특정 장치의 인스턴스를 나타내는 값을 전달할 수 있을 뿐만 아니라 장치의 타입을 구분할 수 있는 값을 전달할 수도 있다.



**(pszName 매개변수의 추가적인 이해를 위해 헷 이용)**

**dwDesiredAcess**: 장치와 데이터를 어떻게 주고받기를 원하는지 결정하기 위해 사용된다.

**dwShareMode**: 장치의 공유 특성을 지정하는 데 사용된다. 이 매개변수는 CreateFile을 이용하여 장치에 대해 열기 작업을 이미 수행한 경우에 추가적으로 동일 장치에 대해 CreateFile을 수행한 경우, 열기 작업을 어떻게 수행할지를 제어하기 위해 사용된다.

**lpSecurityAttributes**: 보안 정보를 설정하거나 CreateFile이 반환하는 핸들을 상속 가능하도록 구성할 것인지의 여부를 결정하는 SECURITY\_ATTRIBUTES 구조체를 가리키는 포인터로 설정된다.

기본 보안 특성을 가지고 있으며, 상속이 불가능한 핸들을 얻으려는 경우에는 단순히 NULL 값을 전달하면 된다.

**dwCreationDisposition**: CreateFile 함수를 파일 장치에 대해 사용할 때 가장 큰 의미를 가진다.

|  |  |
| --- | --- |
| **플래그 값** | **의미** |
| CREATE\_NEW | 새로운 파일을 생성한다. 동일 이름이 존재하면 실패한다. |
| CREATE\_ALWAYS | 기존 동일 이름이 존재 여부를 고려하지 않고 파일을 생성한다. 이미 동일 이름의 파일이 존재하면 덮어쓴다. |
| OPEN\_EXISITING | 기존에 존재하는 파일이나 장치를 연다. 존재하지 않으면 실패한다. |
| OPEN\_ALWAYS | 기존에 동일 이름의 파일일 존재하면 그 파일을 열고, 존재하지 않으면 새로운 파일을 생성한다. |
| TRUNCATE\_EXISTING | 기존에 존재하는 파일을 열고, 그 크기를 0으로 만든다. 파일이 존재하지 않으면 실패한다. |

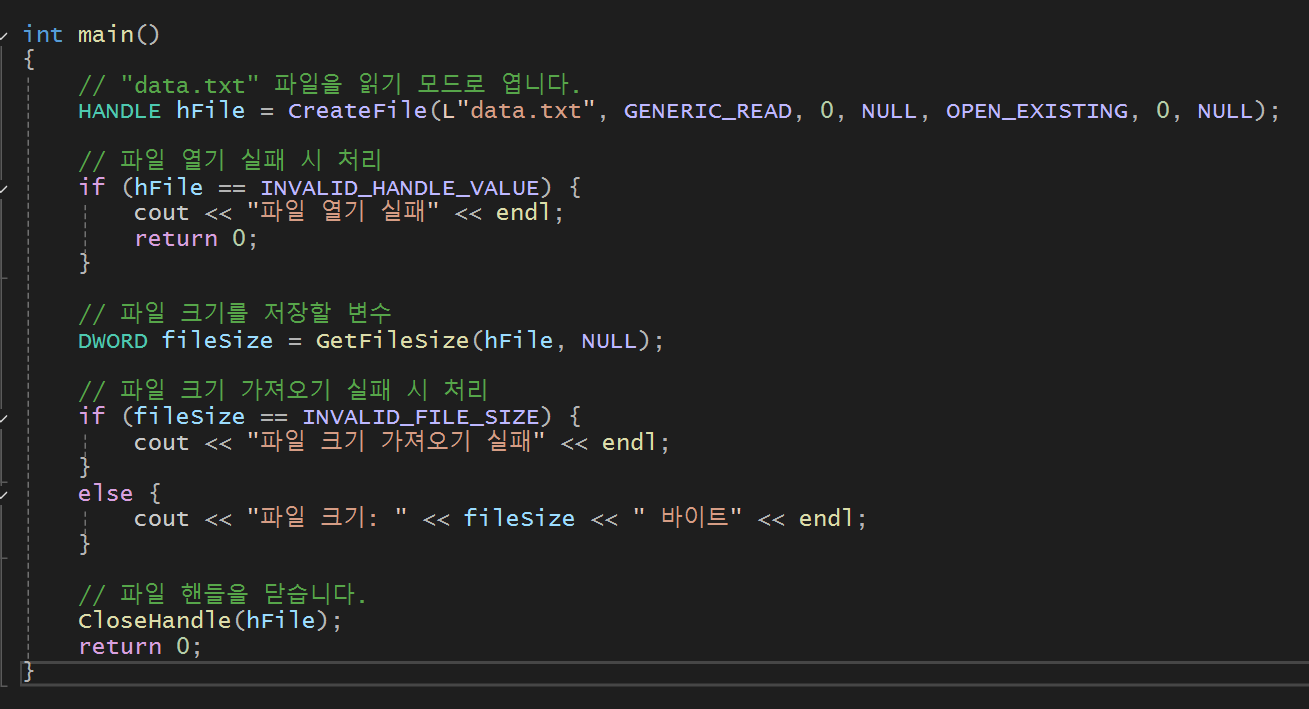
**dwFlagsAndAttributes**: 이 매개변수는 2가지 목적으로 사용된다. 하나는 데이터를 송수신할 때 세부적인 통신 플래그를 설정하기 위한 용도로 사용되며, 다른 하나는 파일의 특성을 설정하기 위한 용도로 사용된다. 통신 플래그를 설정하는 용도로 사용되면 어떤 방식으로 장치에 접근할 것인지에 대한 의도를 설정하는 용도로 사용된다. 이 경우 이 값을 달리하여 애플리케이션이 좀 더 효과적으로 수행될 수 있도록 캐시 알고리즘을 최적화할 수 있다.

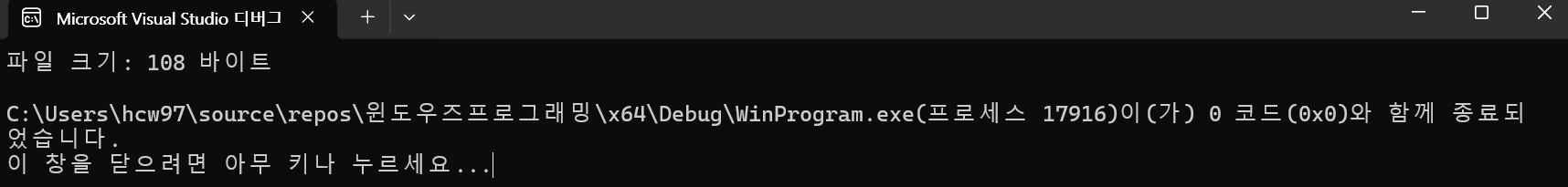
**hFileTemplate**: 이미 열린 파일에 대한 핸들이나 NULL 값을 지정하면 된다.

**파일 장치 이용**

파일 포인터 위치 변경 방법과 파일 크기 변경 방법을 알아볼 것이다.

윈도우는 최초 설계 시부터 파일의 크기를 나타내기 위해 32비트 값이 아닌 64비트 값을 이용하였다. 따라서 이론적으로 파일의 크기는 최대 16EB(엑사바이트)가 될 수 있다.

**파일 크기 얻기**

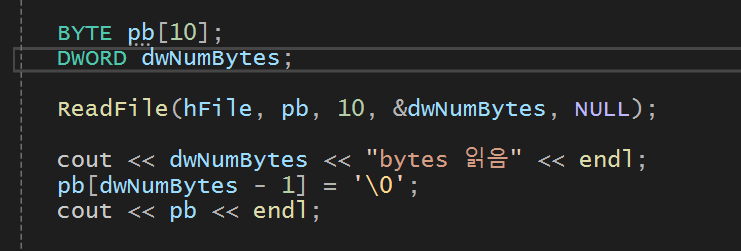


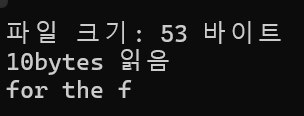
**파일 포인터 위치 지정**

CreateFile을 호출하면 시스템은 파일에 대한 작업을 관리하기 위한 커널 오브젝트를 생성한다.

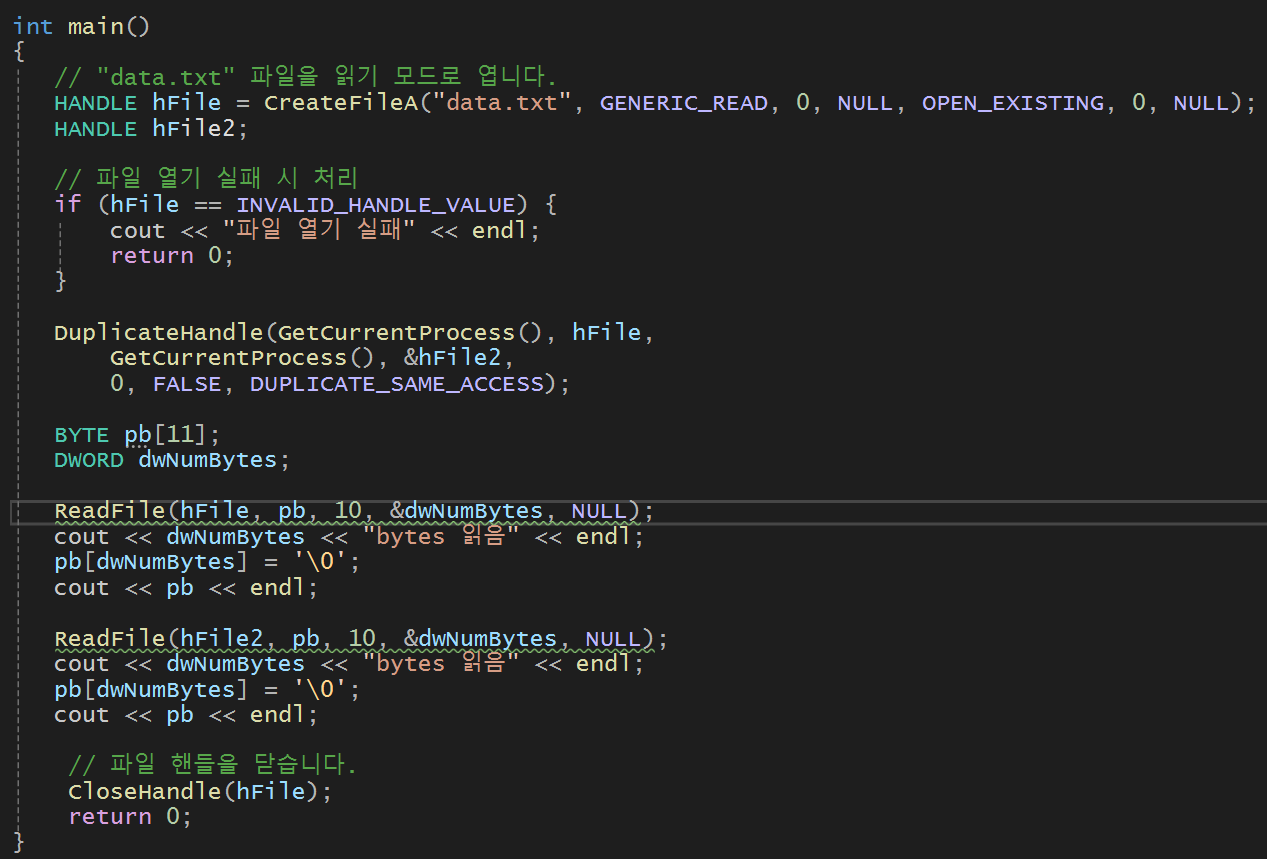
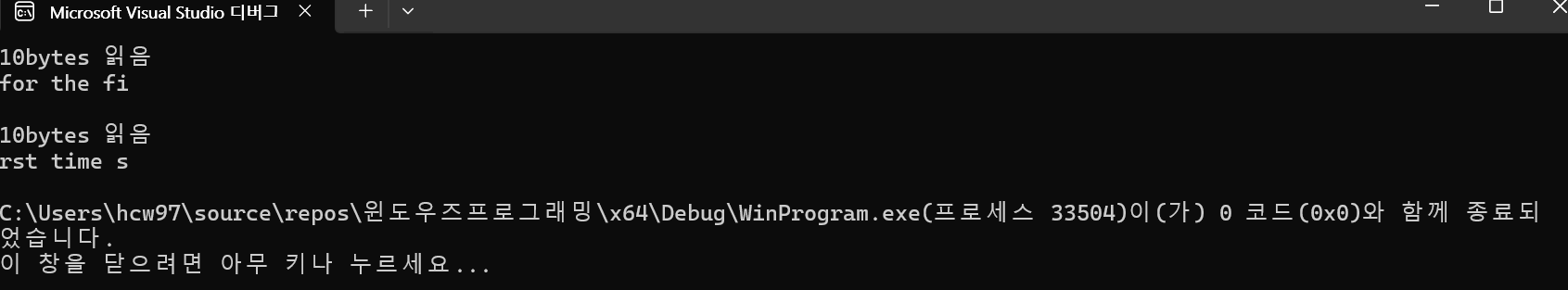
그리고 커널 오브젝트 내부에는 파일 포인터를 가지고 있다. 초기값은 0으로 시작한다.

만약 10바이트를 읽는다면 파일 포인터 값은 10으로 갱신되고, 나중에 다시 ReadFile함수를 호출하여 파일을 읽는다면 10번째 바이트(오프셋10) 부터 값을 읽어 온다.



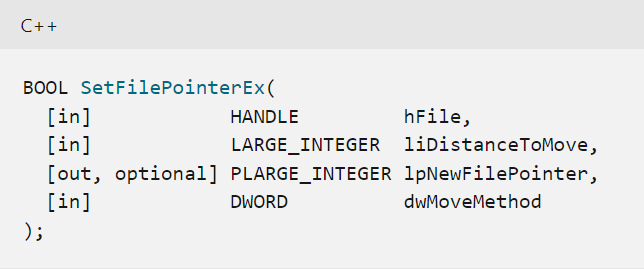


파일에서 10 바이트만 읽어온 모습이다.

각각의 파일 커널 오브젝트는 자신만의 파일 포인터를 가지고 있기 때문에 동일 파일을 여러 번 여는 경우 각각은 서로 독립적으로 수행된다.

하지만 위 코드는 서로 다른 두 개의 파일 핸들이 단일의 커널 오브젝트를 참조하게 만들었다.

둘 중 어떤 핸들을 이용하는 경우라도 동일한 파일 커널 오브젝트 내의 파일 포인터를 갱신하게 되는 것이다.



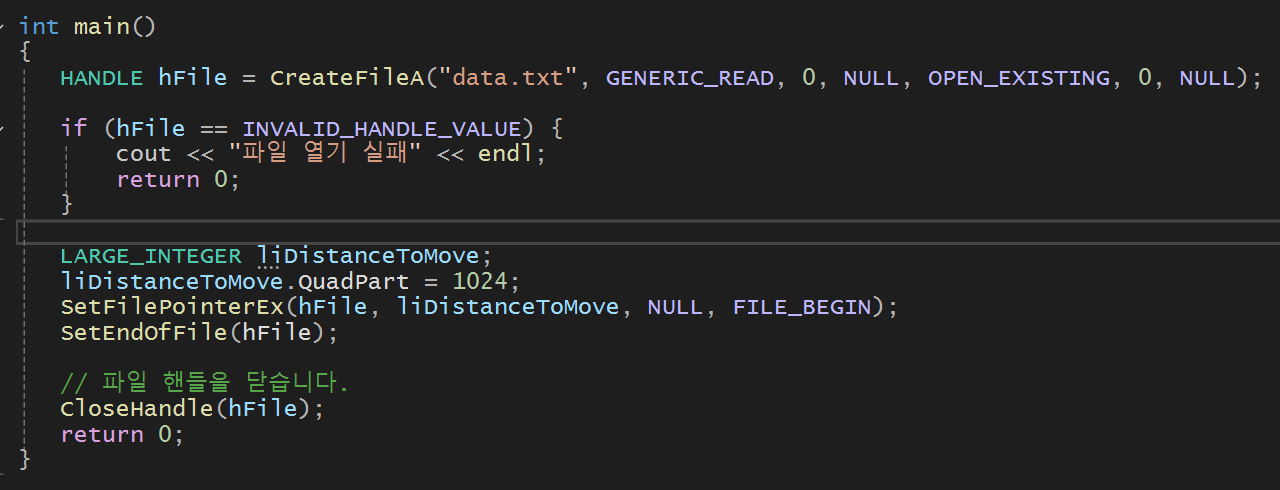
위 함수는 파일의 임의 위치에 접근하려 하는 경우에 사용된다.

liDistanceToMove 매개변수로 파일 포인터를 얼마만큼 이동하고자 하는지를 바이트 단위로 전달하면 된다. 음수를 지정하여 역방향으로도 이동할 수 있다.

**파일의 끝 설정**

일반적으로 시스템은 파일을 닫을 때 파일의 끝을 설정하는 작업을 수행한다. 하지만 떄로는 파일을 닫기 전에 파일을 더 작거나 크게 변경할 필요가 있다. 이러한 작업을 처리하기 위해서는 **SetEndOfFile**을 호출하면 된다.



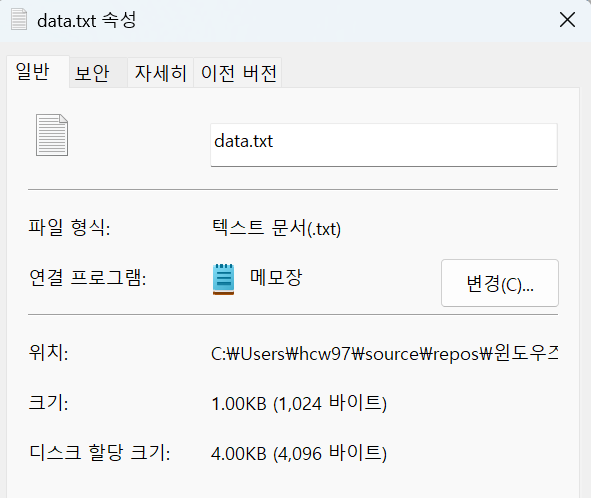


**SetEndOfFile** 함수를 이용하여 파일의 크기를 늘려보는 실습을 진행했다. **하지만 위 코드는 잘 못 된 부분이 있었다.** 그것은 CreateFileA 함수에 플래그 값으로 “**GENERIC\_READ**”를 줬기 때문에 파일의 크기가 바로 늘어나지 않았다. (쓰기 권한은 없었기 떄문)

그래서 해당 부분을



수정하여 코드를 실행했다.



**동기 장치 I/O 수행**

가장 쉽고도 일반적인 동기 장치 I/O 함수는 장치로부터 데이터를 읽고 쓰는 것이며, 이는 ReadFile과 WriteFile 함수를 통해 수행할 수 있다.

**장치로 데이터 플러시하기**

일단 기본적으로 우리가 파일에 어떤 내용을 입력하면 바로 디스크에 내용이 반영되는 것이 아니다. 일반적으로 파일에 데이터를 쓰면, 데이터는 즉시 디스크에 기록되는 것이 아니라 운영 체제의 캐시 버퍼(파일 시스템 캐시)에 먼저 저장된다. 캐시 버퍼를 사용하면 여러 가지 이점이 있다. 특히, 디스크 입출력(I/O) 성능을 최적화하고, 파일 쓰기 작업을 빠르게 수행할 수 있다.

운영 체제는 일정 시간 동안 또는 일정량의 데이터가 쌓일 때까지 데이터를 캐시에 모아두고, 이후 비동기적으로(백그라운드에서) 디스크에 기록한다.

하지만 캐시된 데이터를 장치로 플러시하기 위해 FlushFileBuffers 함수를 사용할 수 있다.

이 함수를 호출하면, 해당 파일 핸들에 대한 모든 캐시된 데이터가 즉시 디스크에 쓰드록 보장된다. 이 과정은 동기적으로 수행되며, FlushFileBuffers 함수는 모든 데이터가 디스크에 안전하게 기록될 때까지 반환되지 않는다.

위 함수를 사용하는 이유는 디스크에 즉시 기록하도록 강제하는 것이므로, 데이터의 일관성을 유지하고 데이터 손실을 방지하는 데 매우 유용하기 때문이다. 이는 특히 중요한 데이터나 크리티컬한 작업**(예: 데이터베이스의 로그 파일 작성 등)**에서 매우 중요한 역할을 한다고 한다.

위 “**파일 캐싱**” 개념을 더 공부하기 위해 아래 링크를 참조했다.

https://learn.microsoft.com/ko-kr/windows/win32/fileio/file-caching

--------------------------------------------------------------------------------------------------------------

진행 상황 유튜브 링크:

|  |  |  |  |
| --- | --- | --- | --- |
| **문제점 정리** |  | | |
| **해결방안** |  | | |
| **다음주차** | 10주차 | **다음기간** | 8. 26 ~ 9. 1 |
| **다음주 할일** | Windows Via C/C++  챕터 10 “동기 및 비동기 장치 I/O” 나머지 | | |
| **지도 교수**  **Comment** |  | | |