작성자(학번 이름) : 2019180052 한창우

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **주차** | 4주차 | **기간** | 2024.7. 15 ~ 21 | **지도교수** |  |
| 이번주 한일 요약 | Windows Via C/C++  챕터 6, 7 “스레드의 기본, 스레드 스케쥴링, 우선순위, 그리고 선호도” | | | | |

<상세 수행내용>

챕터 6. 스레드의 기본

**스레드의 기본**

프로세스는 반드시 하나 이상의 스레드를 가진다. 프로세스는 하나의 프로그램을 실행하는 단위라면, 스레드는 실제로 프로스세 안에서 실제로 작업을 수행하는 실행 단위이다.

즉, 프로세스는 스스로 수행될 수 없으며 단순히 생각한다면 스레드의 저장소로 볼 수 있다.

운영체제가 리소스를 관리하기 위해 커널 오브젝트가 있는 것 처럼 스레드 또한 스레드의 정보를 담고있는 커널 오브젝트가 있다. 시스템은 이를 이용하여 스레드를 관리한다.

스레드는 프로세스 내에서 여러개가 생성될 수 있다. 그렇다면 스레드는 프로세스 주소공간의 데이터들을 공유(스택 제외)하게 된다. 커널 오브젝트 핸들 테이블 또한 스레드별로 존재하는 것이 아닌공유하는 데이터이다. 어떤 문제를 해결하기 위해 자식 프로세스를 생성하여 해결하는 것도 방법이지만, 프로세스를 생성하는 것은 많은 시스템 리소스를 요구하기 때문에 가능하면 비교적 덜 부담되는 스레드를 생성하여 해결하는 것이 좋다. 멀티스레드가 좋을지 멀티프로세스가 좋을지는 상황에 따라 다르므로, 상황에 맞게 적절히 선택하면 된다.

**스레드를 생성해야 하는 경우**

(복습) 프로세스는 실행되면 시스템이 가장 첫번째 스레드인 주 스레드를 만든다.

애플리케이션이 마이크로소프트 C/C++로 컴파일러로 작성된 경우, 주 스레드는 C/C++ 런타임 라이브러리의 시작 코드를 수행하는 것으로 시작하고, 초기화를 마치면 런타임 코드에서 진입점 함수를 호출한다. 그리고 진입점 함수가 끝나고 반환되면 C/C++런타임 코드는 ExitProcess를 호출하고 종료된다.

보통의 작업이라면 주 스레드 하나만으로도 충분하지만, 때에 따라서 스레드를 추가적으로 생성하여 해결하는 것이 효율적일 때가 있다. 아래는 멀티스레드로 처리하여 효율적인 모습의 예시이다.

- 마이크로소프트 Visual Studio IDE는 사용자가 입력을 머출 때, C#과 마이크로소프트 Visual Basic .NET 소스 코드를 자동적으로 컴파일한다. 이를 통해 편집 창 내의 잘못된 코드에 밑줄을 긋고. 마우스 커서를 가져갔을 때 경고나 에러를 보여준다.

- 워드 프로세서 애플리케이션은 페이지 번호 재계산, 철자와 문법 검사, 그리고 프린팅과 같은 작업을 백그라운드로 수행할 수 있다.

- 다른 미디어로의 파일 복사와 같은 작업은 백그라운드로 수행될 수 있다.

- 웹 브라우저는 백그라운드로 서버와 통신을 수행할 수 있다.

이처럼 컴퓨터는 CPU라는 강력한 리소스가 있는데 굳이 유휴상태로 둘 필요가 없는 것이다.

그래서 운영체제와 애플리케이션은 멀티스레드 환경으로 작동되어 효율적으로 일을 처리한다.

멀티스레드를 이용한다는 것은 애플리케이션을 좀 더 확장성이 좋은 구조로 설계할 수 있다.

2개의 CPU를 가지고있는 컴퓨터에서 애플리케이션을 실행할 때, 2개의 스레드를 배치하면 두 개의 CPU를 수행 상태로 유지할 수 있다. 이러면 두 배의 작업을 완료할 수 있을 것이다.

**스레드를 생성하지 말아야 하는 경우**

워드프로세스에서 프린트를 독립적으로 실행하는 스레드를 만든다고 가정하자. 그렇다면 사용자는 프린트 되는동안에도 문서를 작업할 수 있다. 하지만 원본을 프린트하는데 문서 작업을 한다면 내용이 바뀔 것이다.

책에서는 위와 같은 예를 통해, 스레드를 생성하여 문제를 해결하는 것이 항상 좋은 것은 아니며, 공유 자원을 스레드끼리 침범하여 문제가 발생할 수 있음을 보여주고 있다.

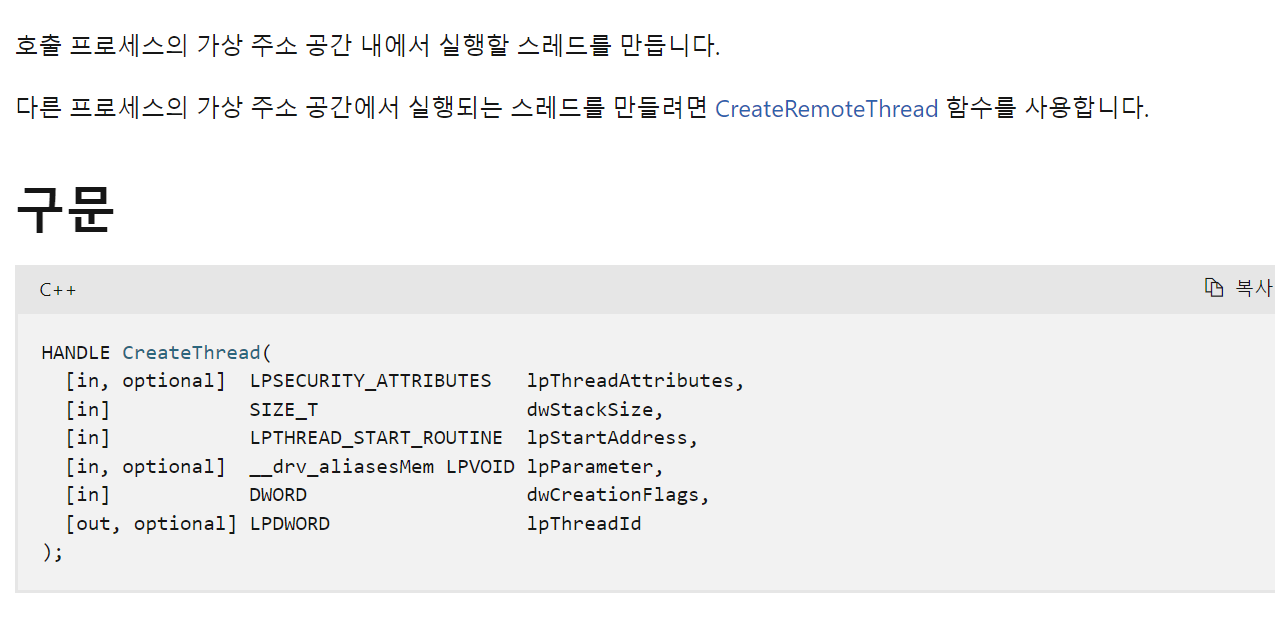
**스레드 함수**

주 스레드의 진입점 함수인 \_tmain 이나 \_tWinMain 인 것 처럼, 모든 스레드는 진입점 함수를 반드시 가진다. 프로세스 내에 두 번째 스레드를 만들려면 새로 생성되는 스레드는 아래와 같은 형태의 진입점 함수를 반드시 가진다.

”DWORD WINAPI ThreadFunc(PVOID Param)”

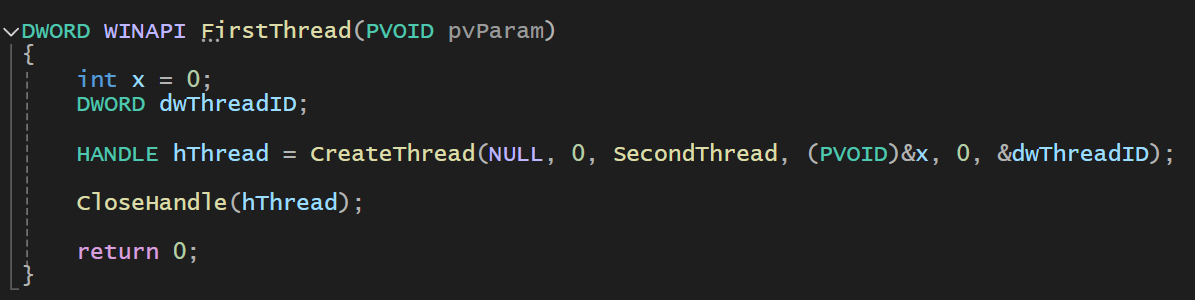
스레드는 소멸되면 반환값을 반환하고 스레드가 상요한테 스택도 반환된다. 또한 스레드의 커널 오브젝트 사용 카운트도 감소된다. 이 값이 0이되면 스레드의 커널 오브젝트는 파괴된다.

**CreateThread 함수**



(새로운 스레드를 생성하는 함수이다. 책에서는 위 함수 대신 C/C++ 코드로 작성되는 경우라면 \_beginthreadex 함수를 사용하라고 한다.)

스레드를 생성하면 시스템은 스레드가 사용할 스택을 확보한다. 새로 생성된 스레드는 스레드를 생성한 프로세스와 동일한 컨텍스트 내에서 수행된다. 따라서 프로세스의 모든 커널 오브젝트 핸들 뿐만 아니라 프로세스에 있는 모든 메모리에 접근이 가능하다. 따라서 동일 프로세스 내의 스레드들은 손쉽게 통신이 가능하다.



CreateThread 매개변수로 &x 라는 인자를 넘긴것은, 해당 변수를 SecondThread 함수의 매개변수로 보낸다는 뜻이다.



위 코드를 작성하고 실습했다.

책에서 FirstThread 함수가 SecondThread 함수보다

먼저 종료되어 스택이 반환된다면, SecondThread 함수가 pvParam을 접근하는 것은 위험할 수 있다는 것을 보여준다.

**스레드의 종료**

스레드는 4가지 방법으로 종료될 수 있다.

- 스레드가 함수가 반환된다. (강력 추천)

- 스레드 함수 내에서 ExitThread 함수를 호출한다. (비추)

- 동일한 프로세스나 다른 프로세스에서 TerminateThread 함수를 호출한다 (비추)

- 스레드가 포함된 프로세스가 종료된다. (비추)

1. 스레드를 종료하려면 스레드 함수가 반환되도록 설계하는 것이 가장 좋다. 그래야 사용한 자원을 적절하게 정리할 수 있기 때문이다. 운영체제가 스레드 스택으로 사용하였던 메모리를 반환해준다.

2. ExitThread함수로 종료한 경우, 스레드를 강제로 종료하고 운영체제가 스레드에서 사용했던 리소스를 정리한다. 하지만 스레드 함수 내에서 생성한 모든 C++ 오브젝트들은 정리되지 않고 남는다.

3. TerminateThread 함수를 사용한 경우, 스레드가 자신이 종료된다는 사실을 알지 못하기 때문에운영체제도 어떠한 리소스도 정리하지 않는다. (아까 실습 때 반환된 스택 프레임 접근은 접근 위 반을 유발했지만 이 경우에는 접근 위반이 발생하지 않는다. 그래서 다른 스레드가 해당 스레드의 스택을 참조한다면 정상적으로 수행될 수 있다.

**프로세스가 종료 vs 스레드가 종료**

저번주(4장)에 알아본 내용 중에 ExitProcess와 TerminateProcess 함수를 호출하는 경우에도 스레드는 종료 된다. 차이점이라면 이러한 함수들을 호출하면 프로세스 내의 모든 스레드가 종료된다는 것이다. 프로세스를 강제 종료하면 프로세스 내에 남아 있는 스레드들에 대해 각각 TerminateThread 함수가 호출된다. 그러므로 이렇게 프로세스를 종료하면 리소스 정리나 디스크로 자료를 저장하는 등의 적절한 정리 작업도 수행되지 못한다.

애플리케이션의 진입점 함수가 반환되면 C/C++런타임 라이브러리의 시작 코드는 ExitProcess를 호출한다. 만약 여러 스레드가 수행중이었더라면, 주 스레드가 종료되기 전에 각각의 스레드들에 대한 적절한 정리 작업이 수행되어야한다. 그렇지 않으면 수행 중인 다른 스레드들이 갑작스럽게 그리고 조용히 종료될 것이다.

**스레드 내부**

스레드가 생성되면 시스템은 스레드의 정보를 담는 커널 오브젝트를 생성한다.

내부를 살펴보면 아래와 같다.

**SP** : 스레드 스택의 주소를 가리키는 레지스터

**IP** : Thread 함수 호출 시 전달된 함수의 주소

**사용 카운트** : 생명 주기

**정지 카운트** : 플래그로 CREATE\_SUSPEND를 넘겨 받았을 시, 스레드가 생성되고 바로 스케쥴링 되지 않음, 하지만 플래그 값이 없었다면 초기값 1을 0으로 만들고 바로 스케쥴링 된다.

**종료 코드** : 반환값

**시그널 상태** : 초기는 FALSE, 스레드가 종료되면 TRUE로 바뀜

각 스레드는 자신만의 CPU 레지스터 세트를 가지는데, 이를 스레드 컨텍스트라고 부른다.

스레드는 항상 프로세스의 컨텍스 내부에서 수행된다.

위 내부에서 IP 레지스터의 경우, NTDLL.dll 모듈이 익스포트하고 있는 RtlUserThreadStart 라는 (문서화 되지 않은) 함수의 주소를 가리키도록 설정된다. 이 함수는 2개의 매겨변수를 받는 함수이다. 이 함수가 다른 함수에 의해 호출되는 것은 아니다. 마치 호출되어서 2개의 매개 변수를 스레드 스택에 삽입된다고 생각할 수 있지만, 운영체제가 임의로 스레드 스택에 삽입하는 것이다.

새로운 스레드가 RtlUserThreadStart 함수를 호출하면 다음과 같은 작업이 수행된다.

- 시스템은 CreateThread 함수 호출 시에 전달한 pvParam 매개변수로 스레드 함수를 호출한다.

- 스레드 함수가 반환되면RtlUserThreadStart 함수는 스레드 함수가 반환한 값을 인자로 ExitThread함수를 호출한다. 스레드 사용 카운트는 감소되고, 스레드는 수행 종료된다.

- 만일 스레드가 예외를 유발하고 이러한 예외가 처리되지 않으면 RtlUserThreadStart 함수가 설정한 SEH 프레임이 예외를 처리하게 된다. 이때 사용자에게 메세지 박스를 출력하는데, 사용자가 “프로그램 닫기” 를 누르면 RtlUserThreadStart 함수는 ExtiProcess를 호출하여 예외를 유발한 스레드뿐만 아니라 전체 프로세스를 종료 시킨다.

정리하자면, 스레드는 진입점 함수를 반드시 호출하는데, 가장 먼저 RtlUserThreadStart 함수가 실행되고 이 함수가 진입점 함수(스레드 함수)를 호출하게 된다. 그리고 스레드가 정상적으로 종료되어 반환값을 RtlUserThreadStart 함수에게 다시 반환한다. 이게 가능한 이유는 RtlUserThreadStart 함수가 스레드 함수를 호출할 때 복귀 주소를 스택에 삽입해주기 때문이다.

챕터 7. 스레드 스케줄링, 우선순위, 선호도

윈도우는 매 20밀리초 정도마다 모든 스레드 커널 오브젝트 중 스케줄 가능 상태에 있는 스레드 커널 오브젝트를 검색하고, 이 중 하나를 선택하여 스레드 컨텍스트 구조체 내에 저장된 레지스터 값을 CPU 레지스터로 로드한다. 이러한 작업을 “컨텍스트 스위칭” 이라고 한다.

컨텍스트 전환이 일어나서 CPU 시간을 할당 받은 스레드는 프로세스의 주소 공간 내에 위치한 코드를 수행하고 데이터를 사용하게 된다. 그러다 할당 받은 시간이 지나면 멈추고 다른 스레드로 컨텍스트 스위칭이 일어나고 실행되는 것이다. 이것이 계속 반복되는 것이며 시스템이 스케줄링하는 모습이다.

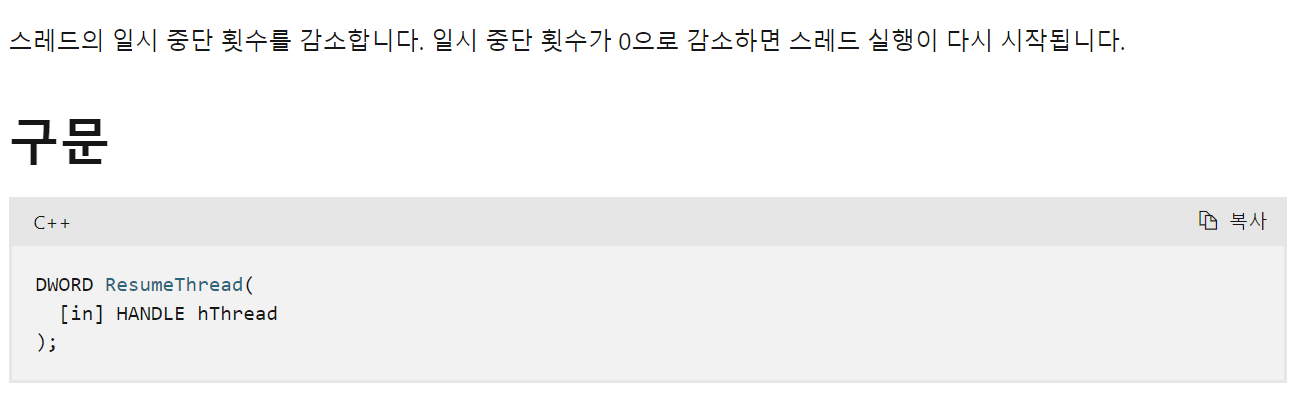
이처럼 윈도우는 언제라도 특정 스레드를 정지하고 다른 스레드를 수행할 수 있기 때문에 “선점형 멀티스레드” 기반 운영체제라고 불린다. 그리고 프로그래머는 이를 개입하여 스케줄링 동작을 제어할 수도 있다.

**스레드의 정지와 계속 수행**

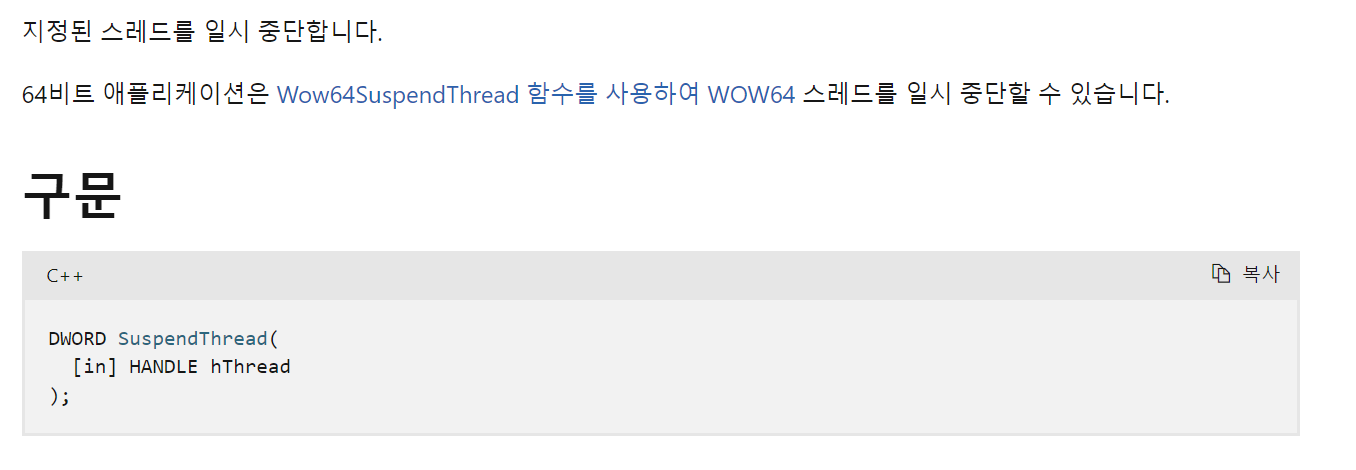
앞장에서 스레드가 만들어질 때 플래그로 CREATE\_SUSPEND를 넘기면 수행을 늦춘다고 하였다.

이는 스케줄링에서 제외 상태를 만드는거였고 운영체제의 관심대상이 아니라는 뜻이다.

이런 스레드를 다시 관심 상태로 바꾸려면, 즉 스케줄링 가능 상태로 만드려면 ResumThread 라는 함수를 사용하면 된다.

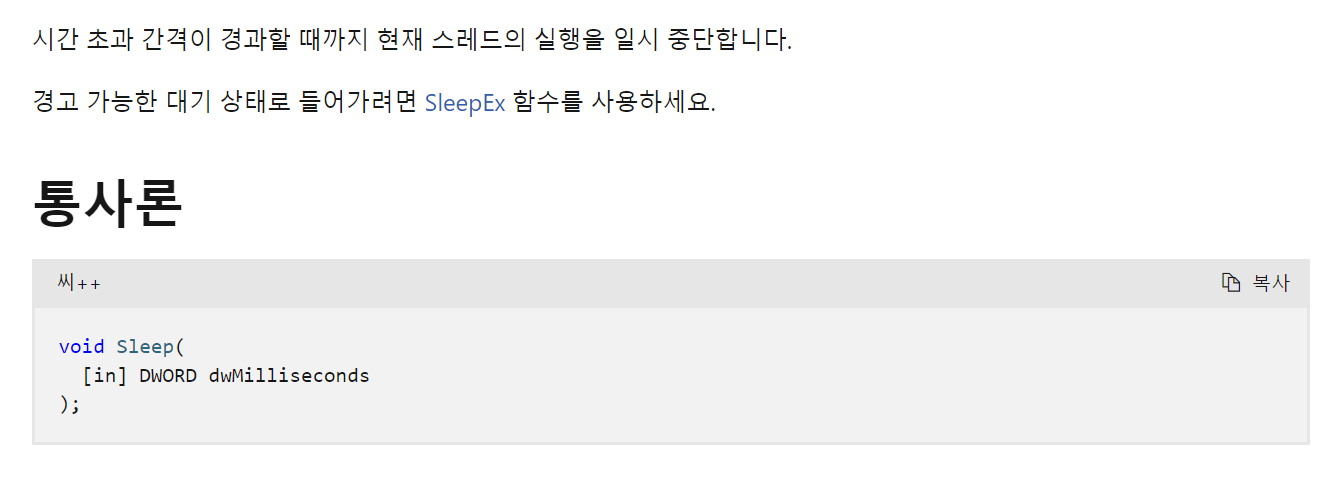


그리고 SuspendThread 라는 함수가 있는데 이는 스레드를 정지하는 함수이다.



주의 사항으로 이 함수가 실행되면 유저 모드에서 수행 중인 코드는 즉시 정지되어 스레드가 수행을 재개할 때 까지 수행되지 않는다는 것이다. 만약 동적할당 중에 멈추게 되면, 해당 스레드는 힙을 잠그게 되며, 이때 다른 스레드에서 힙에 접근하려면 힙을 잠근 스레드가 수행될 때 까지 접근할 수 없게 된다. SuspendThread 함수는 해당 스레드가 어떤 작업을 수행하고 있으며, 스레드가 정지될 때 발생할 수 있는 각종 문제들과 데드락을 피할 수 있는 명확한 방법이 있는 경우에만 사용되어야 한다.

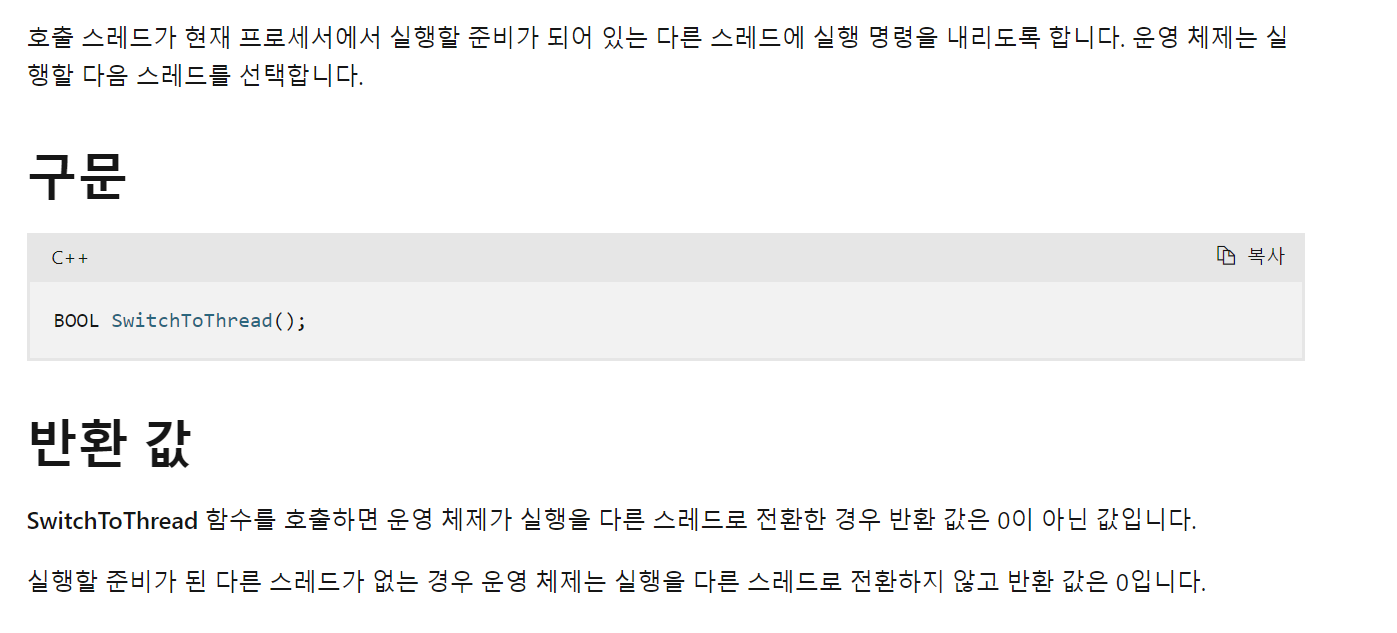
**슬리핑**



스레드 자신을 운영체제에게 스케줄링 되지않도록 명령을 내리는 것이다.

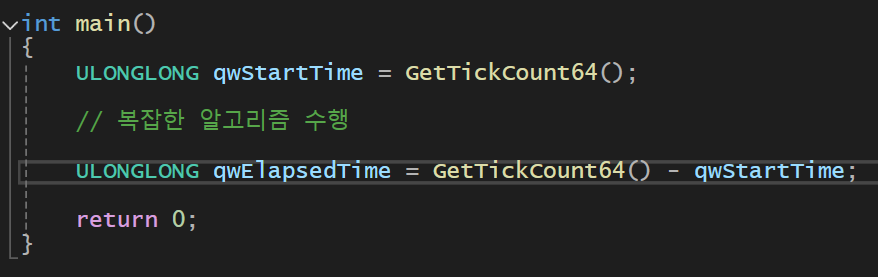
* 자발적으로 타임 슬라이스를 포기한다.
* 100밀리초를 설정하여도, 운영체제의 작업 상황에 따라 100밀리초 이상의 시간이 걸릴 수도 있다.
* INFINITI 인자를 넘겨서 절대 스케쥴되지 못하게 할 수 있다.
* 0을 인자로 넘길 수 있는데, 마찬가지로 남은 타임 슬라이스를 포기하고 잠드는것이다. 하지만 Sleep을 호출한 함수보다 우선순위가 같거나 높은 스케줄링 대상이 없다면 다시 스케줄 될 수도 있다.

**다른 스레드로의 전환**

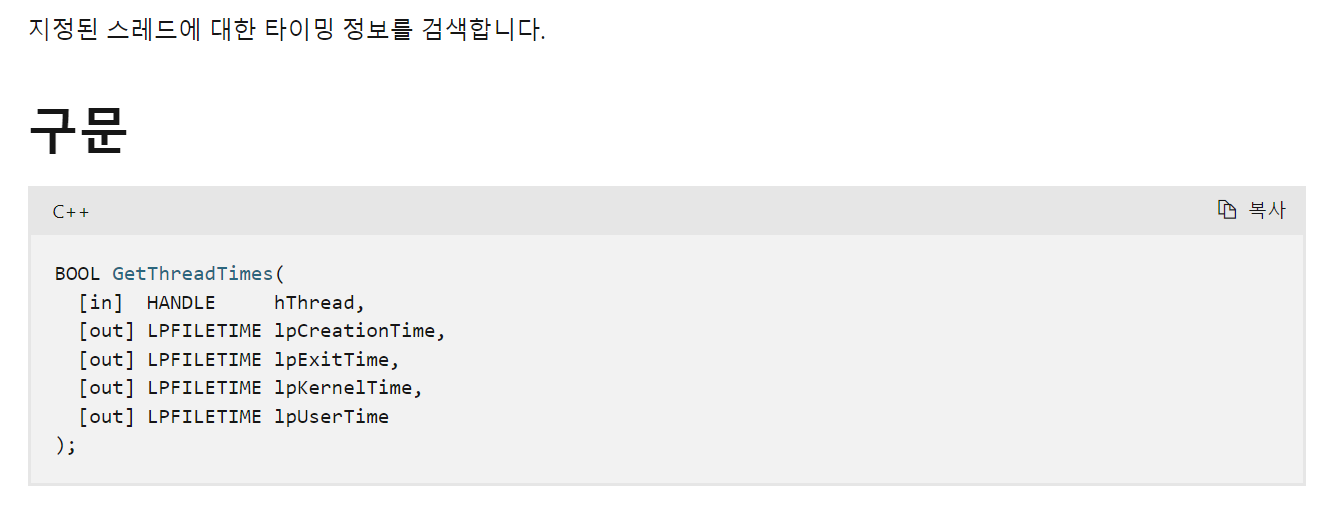


인자로 0을 넣고 Sleep함수를 호출하는 것과 유사하지만, SwitchToThread함수의 경우 호출한 스레드 보다 낮은 우선순위의 스레드들도 수행될 수 있다는 점이다.

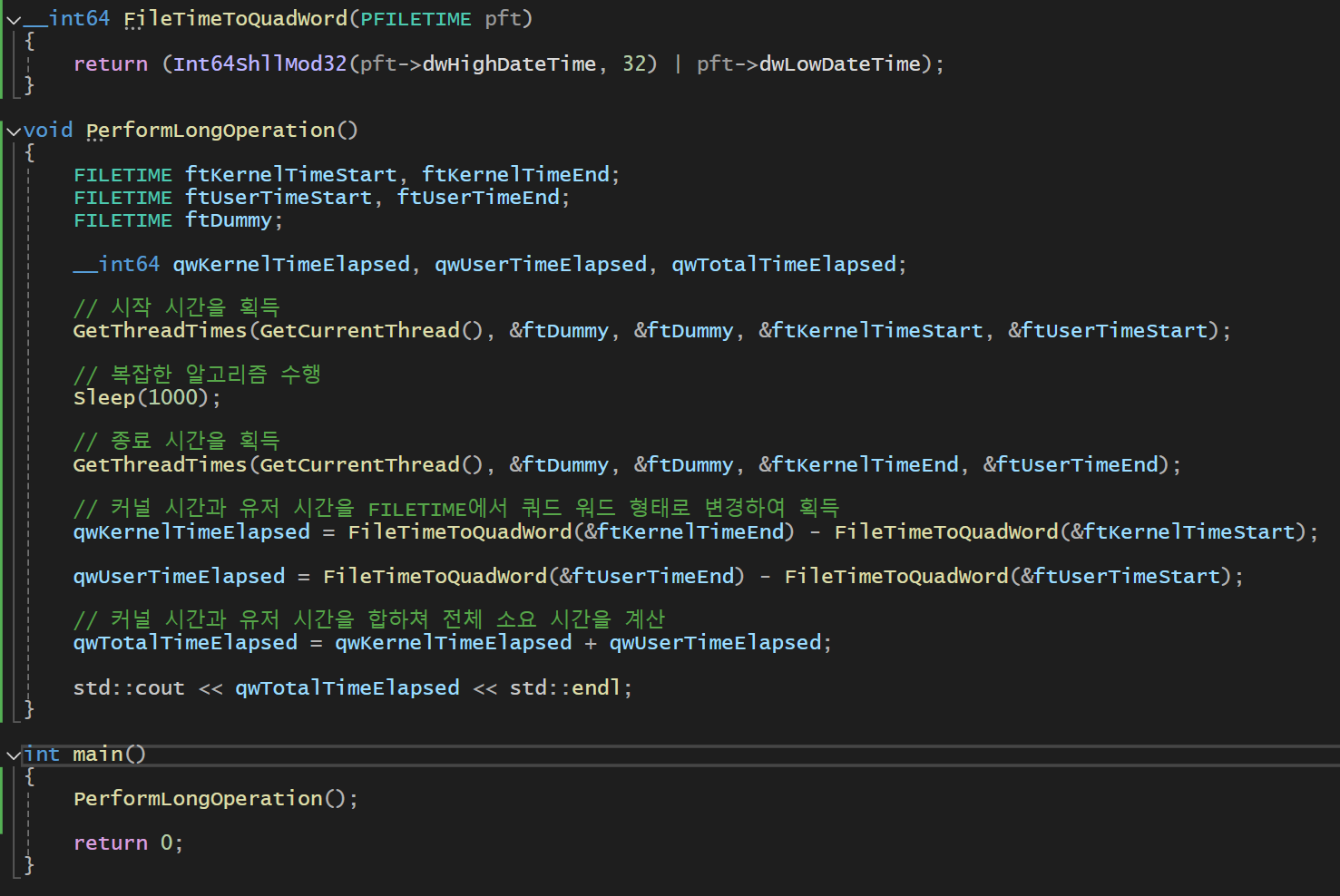
**스레드 수행 시간**



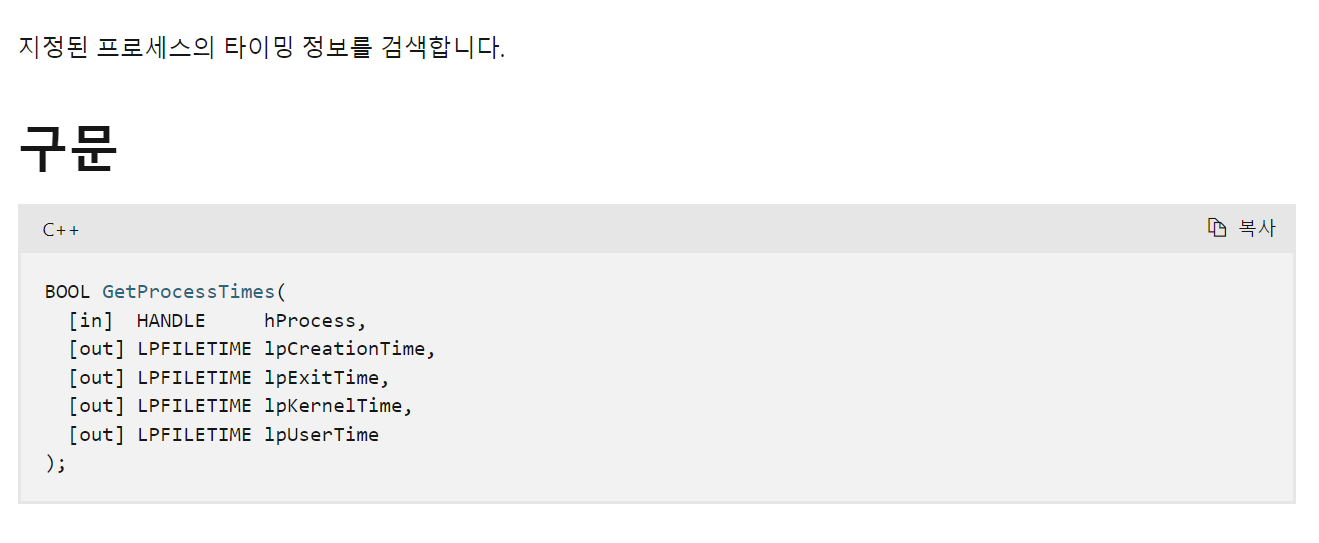
위 코드로는 스레드가 소비한 시간을 정확히 얻기 힘들다. 인터럽트가 발생할 수도 있고, 언제 CPU에 의해서 실행될지도 모르기 때문이다. 하지만 스레드의 정확한 수행시간을 알려주는 함수가 있다.



아래는 GetThreadTime 함수를 사용



(GetThreadTime 함수를 사용)



GetProcessTimes 함수는 위 GetThreadTimes 함수와 유사하며, 프로세스 내의 모든 스레드들이 소요한 시간 정보를 얻어온다.

예전의 운영체제는 클록 타이머를 기반으로 10 ~ 15 단위로 CPU 시간을 계산하였지만, 현재의 운영체제는 머신이 기동된 후부터 얼마만큼의 CPU 사이클이 수행되었는지를 저장하고 있는 64비트 값인 타임스탬프 카운터(TSC)를 이용하여 CPU시간을 계산한다.

스레드가 스케줄러에 의해 정지되면 현재의 TSC 값과 스레드가 재시작되었던 시점에 획득된 TSC값과의 차를 계산한 후 스레드의 수행 시간에 더해준다.

위 코드보다 더 정밀도가 높은 함수를 사용하여 스레드의 소요시간을 구할 수 있는데, 그 함수는

QueryPerformanceFrequency 와 QueryPerformanceCounter 함수 이다.

이 함수들은 윈도우 스케줄러가 해당 스레드를 선점하지 않을 경우에만 정확하게 시간을 측정할 수 있다.

또 다른 정밀도 측정 함수로 C++ 컴파일러가 제공하는 \_\_rdtsc 내장함수가 있다.

\*\*선점(Preemption)\*\*: 운영체제가 현재 실행 중인 스레드를 중지시키고, 다른 스레드를 실행하는 것을 의미합니다. 이 과정은 스레드가 자신의 작업을 완료하기 전에 여러 번 발생할 수 있습니다.

**사이클을 알아오는 함수**

- QueryThreadCycleTime, QueryProcessCycleTime

- 프로세서의 주파수 값을 알아오는 함수. 프로세서의 주파수가 환경에 따라 가변적일 수 있다.

**우선순위**

스레드들은 다양한 우선순위를 가질 수 있다. 모든 스레드들은 0(가장 낮은) ~ 31(가장 높은) 범위 내의 우선순위 번호를 가진다. 시스템은 높은 우선순위를 가진 스케줄 가능한 스레드를 선택하고 라운드 로빈 방식으로 이러한 스레드들을 수행한다. 선택된 스레드는 할당 받은 타임 슬라이스 만큼 실행하고 끝나면, 동급의 우선순위를 가진 스레드를 탐색하여 실행권을 넘겨주게 된다.

우선순위가 높은 스레드가 너무 많은 CPU 시간을 사용해서 낮은 우선순위의 스레드들이 수행되지 못하는 상황을 “기아 상태” 라고 한다. 기아 상태는 비교적 멀티프로세스 환경에서는 적게 발생한다.

애플리케이션 설계 시에는 최종 사용자가 우리가 개발한 애플리케이션 외에도 다른 애플리케이션을 동시에 사용할 가능성이 있는지에 대해 고려해야한다. 우리가 개발한 애플리케이션 내의 스레드가 어느 정도의 응답성이 요구되는지를 판단하고, 이를 기준으로 프로세스의 우선순위 클래스를 결정해야 한다.

**프로세스 우선순위 클래스**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 실시간 | 높음 | 보통 이상 | 보통 | 보통 이하 | 유휴 상태 |

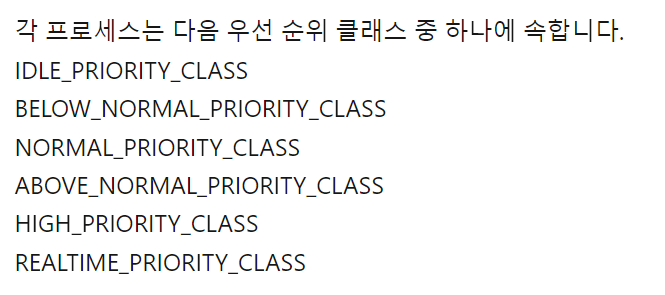
프로세스는 스케줄의 단위가 아니며 스레드만이 스케줄될 수 있다. 프로세스 우선순위 클래스는 마이크로소프트가 스케줄러의 내부적인 동작 방식을 사용자가 직접 다루지 않아도 되도록 하기 위해 만든 추상적인 개념이다.

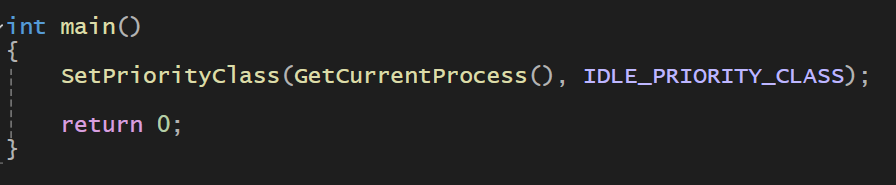
**상대 스레드 우선순위**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 타임 크리티컬 | 가장높음 | 보통 이상 | 보통 | 보통 이하 | 가장 낮음 | 유휴 상태 |

이러한 스레드 우선순위는 프로세스 우선순위에 상대적이다. 앞써 모든 스레드들은 0 ~ 31 범위를 가진다고 하였다. 시스템은 프로세스 우선순위 클래스와 상대 스레드 우선순위를 이용하여 우선순위 레벨로 매핑을 수행한다.

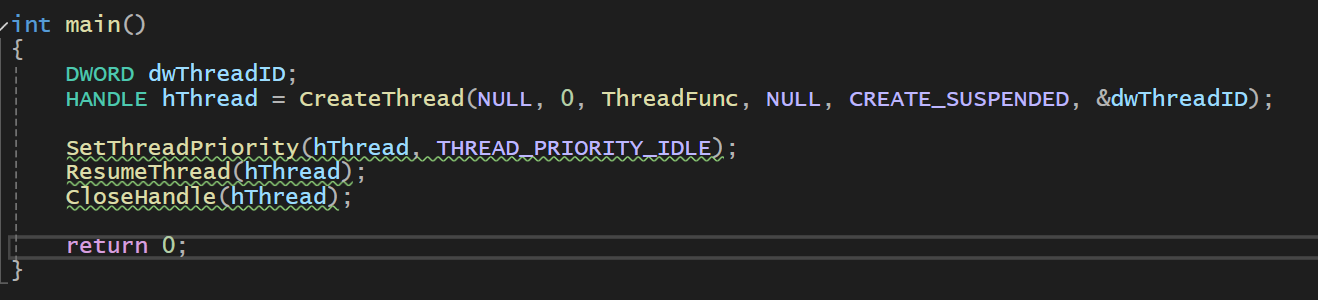
**우선순위 프로그래밍 실습**





위 코드는 해당 프로세스의 우선순위를 유휴 상태로 변경하는 코드이다.

SetPriorityClass 는 위 그림의 구분자 중 하나를 인자로 넘기면서 우선순위를 바꾸는 것이다.



위 코드는 상대 스레드 우선순위를 유휴 상태로 변경하는 코드이다.

CreateThread를 통해 생성된 스레드는 “보통” 등급으로 생성된다. 그래서 CreateThread 함수를 호출할때 매개변수로 CREATE\_SUSPENDED를 넘겨서 해당 스레드가 바로 실행되지 못하게 하고,

SetThreadPriority 함수를 사용하여 유휴상태로 변경한다. 그리고 ResumeThread를 호출하여 실행한다. 이러면 유휴상태 등급의 상대 스레드 우선순위를 가지지만 언젠가는 운영체제의 스케줄러에 의해 실행될 것이다.

진행 상황 유튜브 링크:

|  |  |  |  |
| --- | --- | --- | --- |
| **문제점 정리** |  | | |
| **해결방안** |  | | |
| **다음주차** | 5주차 | **다음기간** | 7. 22~ 7. 28 |
| **다음주 할일** | 5주차에 계획되어있는 내용들 | | |
| **지도 교수**  **Comment** |  | | |