

5. WĄTKI (Thread)

Każdy proces w momencie utworzenia posiada główny wątek oparty o funkcję **WinMain / main**.

Główny wątek programu może tworzyć wątki potomne, a te następne, w wyniku czego powstaje drzewiasta hierarchia wątków, wykonywanych synchronicznie.

- Wątki **nie stanowią** oddzielnego programu.
Są współbieżnie wykonującym się fragmentem kodu danego procesu.
Każdy z nich posiada własną strukturę danych.

Wątek ma dostęp do uchwytów wszystkich obiektów jądra swojego procesu, całej jego pamięci oraz stosów pozostałych wątków należących do tego samego procesu.

W komputerze z jednym CPU w danej chwili może być wykonywana tylko jedna instrukcja, jednego aktualnie aktywnego wątku.

Gdy zostanie przerwana praca wątku system zapamiętuje informacje w specjalnej strukturze o pozycji w kodzie, w której został zatrzymany wątek.

Gdy wątek z powrotem staje się aktywny struktura służy do otworzenia pierwotnego stanu rejestrów procesora i innych kluczowych stanów systemu z chwili przerwania pracy wątku.

➔ **Zakończenie pracy procesu oznacza zakończenie pracy jego wątków.**

- System przydziela każdemu wątkowi **kwant czasu** procesora, o wielkości zależnej od wielu czynników (np. priorytet).

Tworząc wątek wskazujemy **funkcję** jaka ma być jednocześnie wykonywana z innymi wątkami procesu, w zamian otrzymujemy **uchwyt wątku**, który pozwala kontrolować go.

- Funkcja realizowana przez wątek może wywoływać **inne** funkcje programu.

➔ **Dużo zadań można wykonać znacznie wydajniej oraz lepiej bez wątków**

Zastosowanie wątków może być korzystne gdy jakaś operacja jest wykonywana długo, lub gdy jakieś działanie będzie implementowane w tle programu.

Windows co około **20 ms** przegląda wszystkie aktualnie istniejące w jądrze ObiektyWątki.

Niektóre z nich nadają się do wykonania.

System wybiera jeden z tych obiektów i łąduje zawartość jego kontekstu do rejestrów CPU.

Uzyskując dostęp do CPU wątek wznowia wykonanie kodu i operacji na danych w przestrzeni adresowej procesu.

Gdy minie dalsze 20 ms, Windows zapisuje rejestry CPU z powrotem w kontekście wątku i przerywa jego działanie.

System ponownie przegląda wszystkie ObiektyWątki gotowe do wykonania, wybiera jeden z nich, łąduje kontekst do rejestrów CPU i wznowia jego działanie.

Windows rejestruje liczbę przełączeń kontekstu każdego wątku.

Czy można zagwarantować uruchomienie wątku w **określonym** czasie od pewnego zdarzenia **???**

Odpowiedź: **nie można**.

Windows **nie** został zaprojektowany jako SO czasu rzeczywistego.

- ➔ System przydziela CPU tylko tym wątkom, które są zaszeregowane do wykonania.

Większość wątków nie spełnia tego warunku.

Niektóre ObiektyWątki mogą mieć licznik zawieszzeń ustawiony na wartość większą od zera.

Oznacza to, że odpowiadające im wątki są Zawieszone.

Inne wątki nie mogą się wykonywać z powodu oczekiwania na jakieś zdarzenie.

Uruchomiłeś **Notatnik** i nic nie piszesz, wątek **Notatnika** nie ma co robić.

- System **nie przydziela czasu CPU wątkom, które nic nie robią**.

Gdy wprowadzisz do niego tekst, system automatycznie zaszereguje wątek **Notatnika** jako gotowy do Wykonania.

- ➔ **Nie oznacza to natychmiastowego przydziału CPU.**

Trafi do kolejki i **po jakimś** czasie **system** wybierze go do realizacji.

W Windows nie istnieje pojęcie **zawieszania i wznowiania procesów**, gdyż procesy nie mają nigdy przydzielanego czasu CPU.

Jak można zawiesić wszystkie wątki w jakimś procesie ?

System Windows pozwala jednemu procesowi zawiesić wszystkie wątki w drugim procesie, ale musi to być proces wywołujący funkcje:

WaitForDebugEvent i **ContinueDebugEvent**

System Windows nie oferuje innego sposobu zawieszenia wszystkich wątków w procesie ze względu na warunki wyjściu.

W trakcie zawieszania wątków może dojść do utworzenia nowego.

System musi w jakiś sposób zawieszać wszystkie pojawiające się w tym czasie wątki, co zostało wbudowane w mechanizm debugowania systemu.

Definicje pojęć:

Niesygnalizowany stan wątku: wątek istnieje niezależnie od tego czy jest w danym momencie aktywny (wykonywane są jego instrukcje) czy też nie.

Sygnalizowany stan wątku: gdy zakończy pracę.

3.1. Tworzenie wątków funkcjami biblioteki run-time

Funkcja **CreateThread** tworzy w Windows wątki.

Czasami wchodzi w konflikt z niektórymi funkcjami biblioteki czasu wykonania (run-time library) kompilatora.

➔ Pisząc wielowątkowe programy w C++ zaleca się używać funkcji biblioteki run-time:

_beginthread i **_beginthreadex**, deklarację zawiera plik **<process.h>**.

3.1.1. Funkcja _beginthread

unsigned long **_beginthread** (

- ❶ void (**_cdecl** ***start_address**)(void*),
// adres funkcji bazowej, która zostanie uruchomiona w wątku.
- ❷ unsigned **stack_size**,
// rozmiar stosu przydzielony wątkowi.
// wartość zero ustawia wielkość stosu jak dla głównego wątku.
- ❸ void ***arglist**); // adres obiektu, jaki zostanie przekazany do funkcji bazowej wątku

Funkcja bazowa wątku: -nie zwraca wartości;
-pobiera jeden parametr, wskaźnik typu **void**;

Prototyp funkcji bazowej: **VOID Watek**(LPVOID **parm**)

- Funkcja zwraca **uchwyt** (wskaźnik) wątku, jeżeli utworzenie wątku **powiedzie się**, zaś w przypadku **niepowodzenia** zwraca wartość **-1** (0xFFFFFFFF).

_cdecl -metoda wywołania funkcji, powodująca odkładanie parametrów na stosie w odwrotnej kolejności niż są przekazywane do funkcji.

Funkcja o metodzie **_cdecl** nie zdejmuje swoich parametrów przed zakończeniem ze stosu, robi to funkcja wywołująca; kod generowany jest nieco większy; umożliwia obsługiwanie funkcji o zmiennej liczbie parametrów.

_stdcall -metoda wywołania sama zdejmuje swoje parametry ze stosu, informacja o ich liczbie i rozmiarze zapisana jest na stałe w kodzie funkcji.

Uchwyt zapewnia dostęp do wątku, można przekazywać go do dowolnej funkcji pracującej z wątkami.

Jest ważny do momentu zamknięcia, można go używać po zakończeniu pracy wątku.

Uchwyt wątku typu **unsigned** można zastąpić typem **HANDLE**.

(zdefiniowany w pliku nagłówkowym <Winnt.h>, jako: **typedef void *HANDLE**;

Należy wówczas wartość zwracaną przez funkcję **_beginthread** i **_beginthreadex** jawnie rzutować na typ **HANDLE**; inaczej kompilator zgłosi błąd.

(**HANDLE**) **_beginthread**(...)

Wątki utworzone przez **_beginthread** powinny wykonywać **niezależne** zadania.

Gdy kończy się działanie wątku funkcja **_endthread** automatycznie zamyka ich uchwyt (niejawnie wywołanie **CloseHandle**).

Obiekt Wątku jest niszczone **zanim** przejdzie w stan **sygnalizowania**, zatem nie można wykorzystać go do operacji synchronizacji (**nie można oczekiwać na zakończenie pracy wątku**).

➔ Nie stosować funkcji **_beginthread**, gdy wątki będą synchronizowane.

3.1.2. Funkcja _beginthreadex

unsigned long **_beginthreadex** (

- ❶ void ***security**,
// adres do struktury opisującej atrybuty
- ❷ unsigned **stack_size**,
// jak dla funkcji **_beginthread**
- ❸ unsigned (**_stdcall** ***start_address**)(void*)
- ❹ void ***arglist**,
// jak dla funkcji **_beginthread**
- ❺ unsigned **initflag**,
- ❻ unsigned ***thrdaddr**

);

start_address -adres funkcji, która będzie wykonywana jako nowy wątek programu.

Zwracana przez nią wartość jest typu unsigned -równoznaczny z UINT.

Prototyp funkcji bazowej: **UINT WINAPI Watek**(LPVOID **parm**)

initflag -początkowy stan wątku:

-dla wartości zero zacznie się on wykonywać od momentu utworzenia,

-dla wartości **CREATE_SUSPENDED** (0x00000004) pozostanie zawieszony do momentu jawnego uruchomienia.

thrdaddr -adres zmiennej typu unsigned, w której zostanie zapisany identyfikator wątku.

Jeżeli utworzenie wątku **powiedzie się** zwracaną wartością jest jego **uchwyt** w innym razie wartość **zero** (0) (**nie -1** jak dla funkcji **_beginthread**).

3.1.3. Funkcja realizowana przez wątek (funkcja bazowa wątku)

Składnia funkcji, której adres przekazujemy do **_beginthread** lub **_beginthreadex**:

void **_cdecl** **Thread**_(void* **lpVoid**); // dla **_beginthread**

unsigned **_stdcall** **Thread**_(void* **lpVoid**) // dla **_beginthreadex**

```
VOID Funkcja_Watkowa(LPVOID jeden_parametr)
{
    // blok wyłuskujący z jeden_parametr: zmienne n ora z wsp
    int i, k;
    for (i = 0; i < n; i++) {
        X[i] = pow(sin(i), 6) / wsp; // X[] -tablica globalna
        if (i == 55) _endthread( );
    }
    for (k = 0; k <= 55; k++) s += X[k];
    _endthread( ); // jawne zakończenie - opcynie może nie być
}
```

MS Visual C++ domyślnie wywołuje funkcje jako **_cdecl**.

Dołączając **<windows.h>** można: **void*** zastąpić **LPVOID**;
unsigned zamienić na **UNIT**;

➤ Konwencję wywołania **_stdcall** zastępuje identyfikator **WINAPI**.

VOID Thread_(LPVOID **lpVoid**) // dla **_beginthread**

UINT WINAPI Thread_(LPVOID **lpVoid**) // dla **_beginthreadex**

Program jednowątkowy **Watek1**.

Programu główny **main** wyświetla 30 liter **A**, wątek tego programu wyświetla 30 liter **Z**.

Odstęp czasu pomiędzy pojawieniem się liter **A** wynosi 40 ms, zaś liter **Z** 20 ms.

Na ekranie wyświetlane będą początkowo w miarę naprzemiennie litery **A** i **Z**, a pod koniec pracy programu powinny być wyświetlane tylko same litery **A**.

➤ Wątek główny i pochodny pracują przemiennie, przerywaniem ich pracy zajmuje się system.

Parametr **parm** w funkcji **Znak** wykorzystano do przekazania zmiennej typu char (reprezentujący wypisywany znak), poprzez 3-ci parametr funkcji tworzącej wątek.

➤ Nie można użyć wartości parametru **parm** bezpośrednio, należy rzutować na typ CHAR.

Program główny po wykonaniu **_beginthread** (utworzenie wątku), nie czeka na zakończenie pracy wątku, pracuje dalej, wykonując kolejne swoje instrukcje.

Wywołanie funkcji **_endthread** w funkcji **Znak** **nie jest** konieczne, wątek i tak zakończy swoje działanie w momencie napotkania końca funkcji.

```
#include <process.h>
#include <windows.h>
#include <stdio>
#include <stdlib>
#include <cmath>
using namespace std;
VOID Znak(LPVOID);

int main() // Watek1
{
    char zn = 'Z';
    printf("adres_zn: %p \n", &zn);
    _beginthread(Znak, 0, &zn); // watek potomny

    for(int i = 0; i < 30; i++) { // 30 liter A co 40 ms
        printf("A ");
        fflush(stdin);
        Sleep(40); // fatalny spowalniacz wyświetlanych znaków
        // for (long k=0; k<60000; k++) pow(pow(pow(sin(k), 3.3), 2.2), 1.1); // spowalniacz
    }

    printf("\n Koniec Main"); // getchar();
    return 0;
}

VOID Znak(LPVOID parm) // 30 znakow z parametru zn co 20 ms
{
    printf("adres_parm: %p \n", parm);
    char znak = *((char*)parm); // konwersja parametrów

    for(int i = 0; i < 30; i++) {
        printf("%c ", znak );
        fflush(stdin);
        Sleep(20);
    }
    _endthread(); // można pominac
}
```

adres_zn: 0023FF77
adres_parm: 0023FF77
Z Z A Z Z A Z A Z Z A Z A Z Z A Z A Z Z A Z A Z
Z A Z A Z Z A Z A Z Z A Z A Z Z A Z A Z Z A Z A
Z A A A A A A A A A A
Koniec Main

Jedna funkcja może być wykorzystana do tworzenia wielu wątków.

Mamy zdefiniowaną funkcję **Znak** i chcemy uruchomić trzy wątki oparte na tej funkcji.

Należy trzy razy wywołać funkcję **_beginthread**, przekazując do niej adres funkcji **Znak** i adres odpowiedniego znaku jako parametru.

```
#include <iostream>
#include <windows.h>
#include <process.h>
using namespace std;
VOID Znak(LPVOID);

int main() // Watek2 - wiele wątków i jedna funkcja bazowa
{
    char zn1 = '1', zn2 = '2', zn3 = '3';
    _beginthread(Znak, 0, &zn1);
    _beginthread(Znak, 0, &zn2);
    _beginthread(Znak, 0, &zn3);
    Sleep(2000); // zmieniać wartość parametru Sleep() i obserwować

    for(int i = 0; i < 30; i++) { // 30 liter A co 40 ms
        cout << "A" << ' ';
        cout.flush();
        Sleep(40);
    }

    cout << "\n Koniec Main";
    // cin.get();
    return 0;
}

VOID Znak(LPVOID lpzn)

char znak = *((char*) lpzn);
for(int i = 0; i < 30; i++) {
    cout << znak << ' ';
    cout.flush();
    for (long k=0; k<60000; k++) pow(pow(pow(sin(k), 3.3), 2.2), 1.1 ); //spowalniacz
    // Sleep(20);
}
```

2 3 1 3 1 2 3 2 1 2 3 2 1 2 3 2 1 2 3 2 1 2 3 2 1 2 3 2 1 2 3 A 2
A 1 A A 2 A 3 A 2 A A 1 A 2 A A 3 A 2 A 1 A 2
Koniec Main 1 2 3 2 1 2 3 3 1 3 1 3 1 3 1 3 1 3 1 3 1 3 1 3
1 1 3 1 3 1 3 1

System operacyjny nie tworzy trzech oddzielnych kopii funkcji **Znak()**.

Tworzy trzy strumienie kodu, które mają **jedne** wspólne Źródło w postaci adresu funkcji **Znak**.

System na przemian - dla każdego wątku tworzy odpowiednią strukturę danych, w której zapamiętuje ich stan z danej chwili; informacja ta używana jest do odtworzenia stanu wątku podczas najbliższego jego uaktywnienia.

➔ Wywołanie funkcji **Sleep(2000)** zatrzymuje pracę głównego programu na **2** sek.; jest to konieczne, gdyż należy poczekać aż zakończą działanie wątki pochodne.

Bez tego opóźnienia wydruk znaku **A** byłby zakłócany, ponadto praca wątków mogłaby zostać przerwana w momencie wcześniejszego zakończenia pracy przez główny wątek procesu.

3.1.4. Zakończenie pracy wątku

➤ Wątek kończy działanie (jest zabijany), kiedy jego funkcja skończy pracę.

Jeżeli funkcje `_endthread` oraz `_endthreadex` **jawnie** kończą pracę wątku w wybranym momencie; automatycznie **zamykany** jest uchwyt wątku.

```
void _endthread(void); // wątek utworzony przez _beginthread  
void _endthreadex(unsigned retval); // wątek utworzony przez _beginthreadex  
    retval -wartość zwracana przez wątek do systemu (kod wyjścia wątku).
```

Funkcja `Thread_` przedstawiona wcześniej zakończy swoje działanie gdy zmienna `i` osiągnie wartość **55**.

Równocześnie zostanie zamknięty uchwyt wątku.

Jeżeli wywołamy funkcję `_endthread` przed ciałem funkcji bazowej to praca wątku zostanie zakończona, zanim zdąży on cokolwiek zrobić.

➔ Funkcja `_endthread` i `_endthreadex` może być wywołana tylko z wnętrza funkcji wątku.

❑ Funkcja `CloseHandle` zamyka watek jeżeli funkcja bazowa wątku nie zamknęła uchwytu wątku.

```
BOOL CloseHandle(HANDLE hObject);
```

hObject -uchwyt obiektu zamykanego; zamknięty uchwyt jest unieważniany.

Funkcja zwraca wartość niezerową jeżeli wywołanie powiedzie się, w przypadku błędu zero.

❑ Funkcji `TerminateThread` **kończy** pracę danego wątku z dowolnego miejsca programu.

```
BOOL TerminateThread( HANDLE hThread, DWORD dwExitCode);
```

hThread -uchwyt wątku, którego pracę chcemy zakończyć;

dwExitCode - kod wyjścia.

Funkcja zwraca wartość niezerową jeżeli wywołanie powiedzie się, w przypadku błędu zero.

Jeden wątek **nie powinien** zamykać drugiego za pomocą funkcji `TeminateThread`; po jej użyciu, system nie usunie stosu zakończonego wątku.

Alternatywa zewnętrznych metod zakończenia pracy wątku:

1. użycie zmiennej globalnej, która przyjmując ustaloną wartość zasygnalizuje, że wątek ma zakończyć swoją pracę.
2. przekazywanie do funkcji wątku adresu struktury, zawierającej pole informujące wątek o momencie zakończenia.

❑ Funkcja `GetExitCodeThread` pobiera kod wyjścia wątku utworzonego za pomocą funkcji `_beginthreadex` zanim zamknięty zostanie jego uchwyt i zniszczony obiekt wątku.

```
BOOL GetExitCodeThread( HANDLE hThread, LPDWORD lpExitCode);
```

hThread -uchwyt wątku, którego kod wyjścia chcemy pobrać;

lpExitCode -adres zmiennej, w której zostanie zapisany kod wyjścia wątku.

Jeżeli wątek **nie** zakończył jeszcze pracy we wskazanej zmiennej zostanie zapisana wartość **STILL_ACTIVE** (0x103).

Funkcja zwraca wartość niezerową jeżeli wywołanie powiedzie się, w przypadku błędu zero.

3.2. Uspianie wątków

VOID **Sleep**(DWORD **dwMS**)

Funkcja zamraza -na okres czasu [ms] wskazany przez parametr- działanie tylko tego wątku w którym została wywołana.

Jeżeli jedynym wątkiem jest funkcja WinMain/main to wywołanie w jej wnętrzu lub z poziomu innej funkcji (z której WinMain korzysta), powoduje zatrzymanie **pracy programu**.

➤ Uśpiony jeden z wątków programu nie wstrzymuje pracy innych wątków.

Wywołanie **Sleep** pozwala wątkowi **dobrowolnie** oddać resztę przyznanego mu czasu CPU.

- System **odsuwa wątek** od wykonania na czas wskazany przez parametr.

Czy wątek Obudzi się o właściwym czasie **zależy w rzeczywistości od reszty wątków**.

- Można podać w parametrze INFINITE.

Powoduje to bezpowrotne odsunięcie wątku od wykonania.

Lepiej jest wyjść z wątku i odzyskać jego stos oraz obiekt jądra.

- Parametr może również przyjąć wartość **0**.

Oznacza to rezygnację wątku z przydzielonego mu czasu CPU.

System przydzieli CPU ponownie temu samemu wątkowi, który wywołał `Sleep(0)`, gdy nie ma żadnych innych wątków o identycznym priorytecie.

3.3. Pobieranie uchwytu wątku

Uchwyt wątku zwraca funkcja `_beginthread` lub `_beginthreadex` podczas jego tworzenia.

Można wartość tą zapisać w zmiennej globalnej i później wykorzystać.

Jest to niewygodne rozwiązanie, gdyż uzależnia pracę wątku od istnienia konkretnej zmiennej.

Wątek podczas pracy może sam określić swój uchwyt.

```
HANDLE GetCurrentThread(VOID);
```

Funkcja pobiera tzw. **pseudo** uchwytu wątku.

Zwraca uchwyt wątku, w którym została wywołana.

➤ Uchwyt daje prawa dostępu do wątku, lecz nie można go używać w innych wątkach.

Nie musi być jawnie zamykany za pomocą `CloseHandle`.

3.4. Zmienne funkcji wątku

W funkcji wątku można:

- korzystać ze zmiennych globalnych (zdefiniowane poza jakąkolwiek funkcją),
- definiować zmienne statyczne funkcji, które będą istniały przez cały okres życia programu,
- korzystać ze zmiennych automatycznych, tworzonych na stosie wątku w trakcie jego pracy.

Zmienne globalne i **statyczne funkcji** są **wspólne** dla **działających wątków**.

Wątki utworzone na bazie dowolnych funkcji mogą modyfikować zmienne globalne.

Wątki utworzone na bazie **jednej** funkcji mogą korzystać ze zmiennych **globalnych** i dodatkowo są dla nich wspólne **statyczne** zmienne lokalne funkcji wątku.

Zmienne automatyczne są prywatne dla każdego wątku.

Tworząc wątek można określić wielkości jego stosu (lub użyta zostanie wartość domyślna).

- **Każdy wątek posiada oddzielny stos**, zatem zmienne automatyczne utworzone w jego obszarze są prywatne dla każdego wątku.

W programie **Watek3a** dwa wątki generują dane i dwa inne sortują wygenerowane dane.

➔ Funkcja **_beginthread** nie daje możliwości synchronizowania pracy wątków.

Zaleca się stosować funkcję **_beginthreadex** co wymaga:

- zmiany **typu** funkcji bazowej na **UINT WINAPI**,
- rzutowania** funkcji tworzącej wątek na typ **HANDLE**.

```
#include<windows.h>
#include<process.h>
#include<iomanip>
#include<iostream>
#include<cmath>
#include<cstdlib>
using namespace std;

double Generuj(float a, float b) {
    double w = (a + (b - a)*(double)rand()/RAND_MAX);
    for (long k= 0; k < 200; k++) log(pow((pow(sin(k) + 1.1,3.3)), 2.2)); // spowalniacz
    return floor(w * 100 + 0.5)/100; }

void WatekGlowny(int, char);
void DispV(int, double *, char *);

UINT WINAPI GenVec(LPVOID);
UINT WINAPI BubbleSort(LPVOID);

int const maxData = 30000;
double A[maxData], B[maxData]; // tablice globalne

struct PARAM {
    int nData;
    double a, b, *X;
    char zn;
};

int main() // Watek3a
{
    HANDLE hWatekG1=NULL, hWatekG2=NULL, hWatekS1=NULL, hWatekS2=NULL;
    UINT ID_G1=0, ID_G2=0, ID_S1=0, ID_S2=0;
    int nData = 30000;
    PARAM parmGA = { nData, 1.1, 9.9, A, 'A' }, parmGB = { nData, 10.1, 19.9, B, 'B' },
    parmSA = { nData, 0, 0, A, 'C' }, parmSB = { nData, 0, 0, B, 'D' };
    DWORD T1 = GetTickCount();
    hWatekG1 = (HANDLE)_beginthreadex(NULL, 0, GenVec, &parmGA, 0, &ID_G1);
    hWatekG2 = (HANDLE)_beginthreadex(NULL, 0, GenVec, &parmGB, 0, &ID_G2);
    hWatekS1 = (HANDLE)_beginthreadex(NULL, 0, BubbleSort, &parmSA, 0, &ID_S1);
    hWatekS2 = (HANDLE)_beginthreadex(NULL, 0, BubbleSort, &parmSB, 0, &ID_S2);

    WatekGlowny(200, '*'); // symuluje obliczenia programu glownego
    cout << "czas = " << GetTickCount() - T1 << endl;
    DispV(0, 48, A, "vektor A:");
    DispV(0, 48, B, "vektor B:");
    cout << "Koniec Programu";
    // cin.get();
    return 0;
} // czy program za każdym uruchomieniem będzie działać poprawnie ?
```

```
void GenVec(int n, double V[ ], double oda, double dob)
{
    for (int i = 0; i < n; i++) V[i] = Generuj(oda, dob);
}
```

```
UINT WINAPI GenVec(LPVOID parametr)
{
    PARAM *parm = (PARAM*)parametr;
    int n = parm->nData;
    double oda = parm->a, dob = parm->b, *X = parm->X;
    char zn = parm->zn;
    for (int i = 0; i < n; i++) {
        X[i] = Generuj(oda, dob);
        if (i%100 == 0) cout << zn;
    }
    X[0] = 10*oda;
}
```

```
void BubbleSort(double *X, int size)
{
    double w;
    for (int i = 1; i < size; i++)
        for (int j = size-1; j >= i; j--)
            if (X[j] < X[j - 1]) { w=X[j-1]; X[j-1]=X[j]; X[j]=w; }
}
```

```
UINT WINAPI BubbleSort(LPVOID parametr)
{
    DWORD T1 = GetTickCount();
    PARAM *parm = (PARAM*)parametr;
    int size = parm->nData;
    double *X = parm->X, w;
    char zn = parm->zn;
    for (int i = 1; i < size; i++) {
        if (i%100 == 0) cout << zn;
        for (int j = size-1; j >= i; j--)
            if (X[j] < X[j - 1]) { w=X[j-1]; X[j-1]=X[j]; X[j]=w; };
    }
    cout << "nczasSort_" << parm->zn << '=' << GetTickCount() - T1 << "mS\n";
}
```

```
void WatekGlowny(int n, char zn)
{
    for (int k1=0; k1 < n; k1++) {
        for (long k=0; k < 99000; k++) log(pow((pow(sin(k)+1.1, 3.3)), 2.2));
        cout << zn << " ";
    } cout << "\n";
}
```



```

ABAAAAABABABCBCCBCBCDCAACDCCDCDCDCDCDCDC*A BCBCBCBABABABABACDCCDCDCDCDCDCDC
DABDADDADADDADADDADADDADADDADADDADADDADADCBDCDCDCDCDCDCDCDCDCDCDCDCDCDCDC
CDDCDDCDDCDDCDDCDBABABABABABABABABABABABABABACDDDDDDDDDDDDDDDDDDDDDDDD
czasSort_D=421ACBmS
CCBCCCBCCCBCCCBCCCBCCCBCCCBCAA* CACCBABABABABABABABACCCCCCCCCCCCCAAAA* AAAAAA
* AAAAAAA* AACBCCCCC
czasSort_C=734mS
AAA* AAAAAA* AAAAAABABABABBB* BBBB* BBBB* BBBB* BBBB* BBBB* BBBB* B* *****
*****
*****
czas = 8906
vektor A:
11 6.06 2.8 8.22 6.25 5.32 4.18 8.98
8.34 7.67 2.63 8.66 7.35 5.62 3.78 1.23
1.9 4.31 2.4 2.56 9.8 5.02 2.15 1.14
1.18 4.43 5.78 6.13 6.4 6.44 2.56 6.93
5.07 4.2 1.6 6.45 7.99 8.16 5.67 3.76
8.81 0 0 0 0 0 0 0
vektor B:
101 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
Koniec Programu

```

```
int main() // Program tradycyjny
{
    .....
    GenVec (nData, A, 1.1, 9.9);
    GenVec(nData, B, 11.1, 99.9);
    BubbleSort(A, nData);
    BubbleSort(B, nData);
    .....
    return 0;
}

void GenVec(int n, double V[ ], double oda, double dob)
{
    for (int i = 0; i < n; i++) V[i] = Generuj(oda, dob);
}

void BubbleSort(double *X, int size)
{
    double w;
    for (int i = 1; i < size; i++)
        for (int j = size-1; j >= i; j--)
            if (X[j] < X[j - 1]) { w=X[j-1]; X[j-1]=X[j]; X[j]=w; }
}
```

3.5. Zawieszanie i aktywacja wątków

Wywołując funkcje inicjalizujące wątek, w jądrze tworzony jest ObiektWątek z licznikiem zawieszzeń ustawionym **wstępnie na 1**, co uniemożliwia natychmiastowe rozpoczęcie wykonania wątku.

Jest to konieczne, gdyż inicjalizacja wątku zajmuje trochę czasu i wątek nie może zacząć działać na niestabilnych strukturach.

Kończąc inicjalizację wątku sprawdza się, czy została użyta flaga **CREATE_SUSPENDED**.

Jeśli tak, następuje powrót z funkcji, a wątek zostaje w stanie zawieszonym.

Jeśli nie, funkcja zmniejsza licznik zawiesznień wątku do **zera**, co oznacza gotowość wątku do wykonania, chyba że czeka on na zdarzenie (np. na dane z klawiatury).

Utworzenie wątku w stanie ZAWIESZONYM umożliwia zmianę jego środowiska (np. priorytetu) jeszcze przed rozpoczęciem wykonywania kodu wątku.

Po zmianie środowiska wątku trzeba dokonać wznowienia jego wykonania.

Realizuje to funkcja **ResumeThread**.

Z każdym wątkiem powiązany jest tzw. **licznik zawieszenia**.

Może osiągnąć wartość **MAXIMUM_SUSPEND_COUNT** (127).

Gdy jego wartość jest **większa od** zera wątek jest traktowany jako **zawieszony**, w innym wypadku jako gotowy do wykonania.

- ❑ Funkcja **ResumeThread** przywraca do pracy zawieszony w działaniu wątek.

DWORD **ResumeThread**(HANDLE *hThread*);

hThread -uchwyt wątku, którego pracę chcemy wznowić.

W przypadku powodzenia wywołania zwracana jest **poprzednia** wartość licznika zawieszenia.

Jeżeli jest ona równa **0** wątek nie był zawieszony.

Jeżeli wynosi **1** oznacza to, że wątek był zawieszony i właśnie został uaktywniony.

Jeżeli jest większa od **1** to wątek wciąż znajduje się w stanie zawieszenia.

Jeżeli wywołanie się nie powiedzie zostanie zwrócona wartość **0xFFFFFFFF**.

Funkcja zmniejsza o jeden wartości licznika zawieszenia wątku, tylko wtedy, gdy jego wartość jest większa od zera.

- Gdy licznik osiągnie zero praca wątku jest wznowiana.

Próba wznowienia pracy działającego wátu **nie modyfikuje** licznika zawieszenia.

- ❑ Funkcja **SuspendThread** umożliwia zawieszenie pracy wątków.

Jeden wątek może być zawieszony wiele razy.

Zamiast używać flagi `CREATE_SUSPENDED` podczas tworzenia wątku, można go zawiesić wywołując funkcję **`SuspendThread`**.

DWORD **SuspendThread**(HANDLE *hThread*);

hThread-uchwyt wątku, który chcemy zawiesić.

Automatycznie zwiększa się o 1 licznik **zawieszenia wątku**.

Przekroczenie dopuszczalnej wartości 127 (zawieszenie wątku ponad 127 razy) daje błąd ale licznik nie będzie zwiększony.

Jeżeli wywołanie się powiedzie funkcja zwraca poprzednią wartość licznika zawieszenia, w innym razie jest to wartość 0xFFFFFFFF.

Każdy wątek może wywołać **SuspendThread** i uśpić inny wątek.

Wątek może **zawiesić** także **siebie**, ale **nie może** siebie wznowić.

- ➔ Należy zachować ostrożność, wywołując funkcję ***SuspendThread***, gdyż aplikacja nie ma pojęcia co robi wątek, który próbuje zawiesić.

Jeśli ten ostatni alokuje pamięć na sterce, oznacza to, że jest w posiadaniu blokady sterty.

Próby dostępu do serty przez inne wątki będą powodowały ich zawieszenie aż do momentu wznowienia pierwszego wątku.

Funkcję ***SuspendThread*** można stosować bezpiecznie, gdy wiadomo co wątek robi w tym momencie.

3.6. Przełączanie na inny wątek

Funkcję **SwitchToThread** pozwala uruchomić inny **gotowy** do wykonania wątek.

```
BOOL SwitchToThread(void);
```

Po wywołaniu **SwitchToThread** SO sprawdza, czy istnieje wątek oczekujący na przydział CPU.

Jeśli nie można uruchomić innego wątku w momencie wywołania *SwitchToThread*, funkcja ta zwraca FALSE, w przeciwnym razie wartość różną od zera.

- Jeśli **nie**, następuje natychmiastowy powrót z funkcji *SwitchToThread*.
- W przeciwnym razie funkcja kieruje do wykonania czekający wątek, który może mieć **niższy priorytet** niż wątek wywołujący *SwitchToThread*.

Nowy wątek dostaje zgodę na działanie przez **jeden kwant** czasu, po czym program szeregujący powraca do normalnego schematu działania.

Pozwala to wątkowi, który potrzebuje jakiegoś zasobu, zmusić wątek o niższym priorytecie -posiadający aktualnie ten zasób- do jego zwolnienia.

Wywołanie **SwitchToThread** podobne jest do wywołania **Sleep(0)**.

SwitchToThread pozwala wykonywać wątki o niższym priorytecie, natomiast **Sleep(0)** natychmiast ustawia wywołujący ją wątek w kolejce do wykonania (wyrzuci on wątek o niższym priorytecie).

Program **Watek4** wykorzystuje funkcje **SuspendThread** i **ResumeThread**

Pierwszy wątek wypisuje literę **A**, a drugi literę **B** w 20-sto ms odstępach.

Naciśnięcie klawisza **A** zawiesz a w ątek wypisuj ący liter ę **A**, je eli pracowa ł w danej chwili, gdy by ł ju ź zawieszony zostanie przywrócony do dzia łania.

Identycznie za pomocą klawisza **B** sterujemy pracą drugiego wątku.

int kbhit(void) -zwracana wartość jest różna od zera, jeżeli w buforze znajduje się znak, umieszczony w wyniku naciśnięcia klawisza klawiatury.

Wartości zero oznacza, że bufor jest pusty (żaden klawisz nie został naciśnięty). Znak można pobrać z bufora funkcja **getch()**.

```
#include <iostream>
#include <windows.h>
#include <process.h>
#include <conio.h>
using namespace std;
VOID ZNAK(LPVOID);

int main() // Watek4 naciśnięcie Spacji kończy Pracę programu
{
    char zn = 0, znA = 'A', znB = 'B';
    HANDLE hWatekA = NULL, hWatekB = NULL;
    BOOL CzyWatekAZawieszony = FALSE;
    BOOL CzyWatekBZawieszony = FALSE;

    // uruchamianie 2-ch watekow i zapamiętanie uchwytu
    hWatekA = (HANDLE)_beginthread(ZNAK, 0, &znA );
    hWatekB = (HANDLE)_beginthread(ZNAK, 0, &znB );

    for(;;) {
        if(kbhit() != 0) { // pobieranie znaku z klawisza
            zn = getch();
            if(zn == 'A' || zn == 'a') // zawieszanie lub wznowianie 1-go wątku
                if(CzyWatekAZawieszony == FALSE) {
                    SuspendThread(hWatekA); // zawieszenie
                    CzyWatekAZawieszony = TRUE;
                } else {
                    ResumeThread(hWatekA); // wznowienie
                    CzyWatekAZawieszony = FALSE;
                }
            if(zn == 'B' || zn == 'b') // zawieszanie lub wznowianie 2-go wątku
                if(CzyWatekBZawieszony == FALSE) {
                    SuspendThread(hWatekB);
                    CzyWatekBZawieszony = TRUE;
                } else {
                    ResumeThread(hWatekB);
                    CzyWatekBZawieszony = FALSE;
                }
        }
        // spacja przerywa pętlę for i kończy funkcję main, koniec pracy wątków
        if(zn == ' ') break;
    }
    return 0;
}

VOID ZNAK(LPVOID lpzn)
{
    for(;;) {
        cout << *((char*) lpzn) << ' ';
        cout.flush();
        Sleep(20);
    }
}
```

3.6. Funkcja SetThreadPriority

Każdy wątek ma przypisany priorytet, będący liczbą z przedziału od **0** (najniższy) do **31** (najwyższy).

W Windows CPU może wykonywać jeden wątek przez około 20 ms, po czym program szeregujący przydzieli CPU następnemu wątkowi (o takim samym priorytecie) gotowemu do wykonania.

System szuka najpierw wątków o priorytecie **31** i przydziela im CPU metodą *round-robin* (karuzelowa).

- ◀ Dopóki istnieją **gotowe do wykonania** wątki o priorytecie **31**, system nigdy nie przydzieli CPU wątkowi o priorytecie zawartym między 0 a 30.

```
BOOL SetThreadPriority( HANDLE hThread,    // uchwyt wątku
                      INT nPriority      // identyfikatorów poziom priorytetu wątku
                      );
```

Funkcja zwraca wartość różną od zera, jeżeli została zakończona sukcesem.

Funkcja zwraca wartość 0 (zero) jeżeli wystąpił błąd.

Dodatkowe informacje o błędzie dostarcza wywołanie funkcji *GetLastError*.

❑ **Priorytety wątków**, ustawiane są względem klasy priorytetu procesu:

THREAD_PRIORITY_IDLE	// Niski, priorytet 16 dla klasy rzeczywistej
THREAD_PRIORITY_LOWEST	// Najgorszy, 2 punkty poniżej normalnego
THREAD_PRIORITY_BELOW_NORMAL	// Poniżej normalnego, 1 punkt niżej niż normalny
THREAD_PRIORITY_NORMAL	// Normalny, zwykły priorytet klasy jego procesu
THREAD_PRIORITY_ABOVE_NORMAL	// Powyżej normalnego, 1 punkt niżej niż normalny
THREAD_PRIORITY_HIGHEST	// Najlepszy, 2 punkty powyżej normalnego
THREAD_PRIORITY_TIME_CRITICAL	// Krytyczny, priorytet 31 dla klasy rzeczywistej

- ◀ Funkcja *CreateThread* tworzy nowy wątek zawsze z **priorytetem normalnym**.

Uwaga. Jeśli wykonuje się np. wątek o priorytecie **6** i system wykryje **gotowy do wykonania** wątek o wyższym priorytecie, natychmiast zawiesi ten pierwszy (nawet, jeśli będzie on w środku swojego kwantu czasu) i przydzieli CPU drugiemu wątkowi pełny kwant czasu.

System tworzy specjalny wątek - **wątek zerowania stron** z priorytetem **0**, który zeruje wszystkie wolne strony RAM i wykonuje się tylko wtedy, gdy w systemie działa inny wątek.

Każdy Proces ma przypisaną pewną klasę priorytetu.

Wątki mają priorytety ustawione względem Procesu, który je stworzył.

- ◀ Zatem jakie numery z zakresu (0 ÷ 31) otrzymują wątki ?

Twórcy aplikacji nie korzystają z numerów bezwzględnych, system sam mapuje klasy priorytetów procesów i względne priorytety wątków na wartości numeryczne.

Większość procesów należy do klasy **Normalny** i ma „**Normalne**” wątki.

Normalny wątek Normalnego procesu otrzymuje priorytet **8**.

- ➔ Zmieniając **klasę** priorytetu **Procesu**, względny priorytet jego **wątku** pozostanie taki sam, natomiast sam **priorytet ulegnie zmianie**.

„**Normalny**” wątek w **Procesie** o „**Wysokim**” priorytecie będzie miał priorytet **13**; zmiana klasy priorytetu **Procesu** na „**Niski**”, zmieni poziom priorytetu **wątku** na **4**.

Identyfikator klas priorytetu PROSESU

Czasu rzeczywistego	REALTIME_PRIORITY_CLASS
Wysoki	HIGH_PRIORITY_CLASS
Powyżej normalnego	ABOVE_NORMAL_PRIORITY_CLASS
Normalny	NORMAL_PRIORITY_CLASS
Poniżej normalnego	BELOW_NORMAL_PRIORITY_CLASS
Niski	IDLE_PRIORITY_CLASS

Parametr: **6** **DWORD dwCreate** funkcji **CreateProcess()**, pozwala ustawić klasę priorytetu **Procesu** wg powyższej tabeli.

```
CreateProcess(↑,↑,↑,↑,↑, HIGH_PRIORITY_CLASS , ↑,.....);
```

Ustawione priorytety procesu, narzucają priorytety wątków w danym procesie w stosunku do wątków z innych procesów.

Względny priorytet wątku	Klasa priorytetu Procesu					
	Niski	Poniżej Norm.	Norm.	Powyżej Norm.	Wysoki	Czasu rzecz.
Krytyczny	15	15	15	15	15	31
Najlepszy	6	8	10	12	15	26
Powyżej normalnego	5	7	9	11	14	25
Normalny	4	6	8	10	13	24
Poniżej normalnego	3	5	7	9	12	23

Priorytety niedostępne dla aplikacji wykonywanych w trybie użytkownika:

17, 18, 19, 20, 21, 27, 28, 29, 30.

Sterownik urządzenia działający w trybie jądra, może używać niedostępnych priorytetów.

Wątek procesu czasu rzeczywistego nie może mieć priorytetu na poziomie niższym niż 16.

UWAGA. To nie **procesy** są szeregowane, lecz **wątki**.

Microsoft stworzył abstrakcyjne pojęcie **klasa priorytetu procesu** w celu odizolowania programistów od wewnętrznego mechanizmu programu szeregującego.

Proces potomny może w trakcie działania zmienić swoją klasę priorytetu, wywołując funkcję:

```
BOOL SetPriorityClass( HANDLE hProcess, DWORD fdwPriority );
hProcess -uchwyt procesu, którego klasa priorytetu ma być zmieniona,
fdwPriority -nowa wartość klasy priorytetu.
```

Przykład zmiany przez proces swojej klasy priorytetu na niski:

```
SetPriorityClass( GetCurrentProcess(), IDLE_PRIORITY_CLASS );
```

Funkcja sprawdzająca **klasę** priorytetu procesu:

```
DWORD GetPriorityClass( HANDLE hProcess );
```

- ➔ Windows **nie** udostępnia funkcji, sprawdzającej **wartość** priorytetu **PROCESU**.

Sprawdzenie względnego priorytetu wątku: `int GetThreadPriority(HANDLE hThread);`

- ◀ Funkcja *CreateThread* tworzy nowy wątek zawsze z **Normalnym względnym** priorytetem.

Program uruchomiony za pomocą interpretera poleceń ma na początku ustawiony priorytet normalny. Program uruchomiony z polecenia Start, może używać klucza wyznaczającego początkowy priorytet.

C:\>START /LOW CALC.EXE

Dozwolone są klucze: /BELOWNORMAL, /NORMAL, /ABOVENORMAL, /HIGH i /REALTIME.

➤ Aby samemu ustawić priorytet wątku, trzeba odpowiednio wywołać funkcję **SetThreadPriority**:

Flaga **CREATE_SUSPENDED** blokuje wykonanie kodu wątku, co pozwala funkcji **SetThreadPriority** ustawić żądany priorytet.

Następnie wywołać **ResumeThread**, aby zaszeregować wątek do kolejki CPU.

Kiedy odblokowany wątek zacznie się wykonywać nigdy nie wiadomo, ale program szeregujący weźmie już pod uwagę nowy priorytet względny.

```
DWORD ID;
HANDLE hThread = CreateThread(NULL, 0, Func, NULL, CREATE_SUSPENDED, ID);

SetThreadPriority(hThread, THREAD_PRIORITY_IDLE);
ResumeThread(hThread); // wznowienie wątku

{ FRAGMENT KODU WĄTKU MACIERZYSTEGO }

CloseHandle(hThread); // niszczy odpowiadający wątkowi Obiekt jądra
```

Program **Watek5** nadaje wątkowi drukującemu znak **'2'** wysoki priorytet względny.

Priorytety **standardowe**

```
*2 1 3 2 *1 3 2 *3 1 **2 3 *1 2 *3 1 2 *3 *1 **3 2 *1 *3 2 1 *3 2 1 *1 2 3 ***1 3 2 *3 1 **2 3
1 ***2 3 *1 2 *3 1 2 3 1 **2 1 *3 *2 *3 *1 **2 3 **1 *2 3 *1 2 3 1 *2 3 1 *2 *3 *1 ***2 1 3 2
*1 3 **2 1 *3 2 *1 3 2 *1 3 *2 1 3 **1 2 3 *1 *2 3 1 *2 3 1 2 *3 1 ***2 3 1 *2 1 3 *2 1 3 2 *1
3 2 ***1 3 *2 *1 3 2 *1 *3 2 1 **3 *2 1 *3 2 1 3 **1 **
```

Priorytet **HIGHEST** dla **zn2 = 2** i **LOWEST** dla pozostałych

```
*2 1 2 3 2 1 2 3 2 2 1 *2 3 *2 ***2 *2 *2 *2 **2 3 *1 3 2 2 1 3 2 2 3 1 2 2 1 3 2 2 1 3 2 2 1
3 2 2 1 3 2 2 3 1 2 2 3 1 2 2 3 1 2 2 3 1 2 2 1 3 2 2 1 3 2 2 3 1 2 2 1 3 2 2 1 3 2 2 1 3
2 2 3 1 2 2 3 1 2 2 3 1 2 2 3 1 2 2 1 3 2 2 1 3 ***3 1 *3 *1 ***3 *1 *3 ***1 *3 *3 ***3 *3
*3 ***3 *3 *3 ***3 *3 *3 ***3 *3 *3 ***3 *3 *3 ***3 *3 *3 ***3 *3 *3 ***3 *3 *3 ***3 *3 *3 ***3
```

Priorytet **TIME_CRITICAL** dla **zn2 = 2** i **LOWEST** dla pozostałych

```
2 1 2 3 2 1 2 3 *2 1 3 2 2 1 3 2 2 1 3 2 2 1 3 2 2 1 3 2 2 1 2 *3 2 2 1 2 *3 2 1 2 3 2 1 2 3
2 1 2 3 2 1 2 3 2 1 2 3 2 1 2 3 2 1 2 3 2 1 2 3 2 1 2 3 2 1 2 3 2 1 2 3 2 1 2 3 2 1 2 3 2 1
2 3 2 1 2 3 *1 *2 2 1 2 3 2 1 2 3 2 1 2 3 *****1 3 **1 *3 ***1 *3 *1 ***3 **1 3 ***1 *3 *1 **3
****1 3 **1 *3 ***1 *3 **1 3 *****1 *3 *1 ***3 **1 *3 *1 ***3 *1 *3 **1 ***3 *1 *3 **1 ***3 *1 *1 **
```

```
#include <iostream>
#include <windows.h>
#include <process.h>
#include <cmath>
using namespace std;

#define MAX_THREADS 3
UINT WINAPI ZNAK(LPVOID);
void WatekGlowny(int, int, char);

int main() // Watek5 - priorytety
{
    int i;
    DWORD Th1, Th2, Th3;
    unsigned ID0, ID1, ID2;
    HANDLE hW[ MAX_THREADS ];
    char zn1 = '1', zn2 = '2', zn3 = '3';
    hW[0] = (HANDLE)_beginthreadex( NULL, 0, ZNAK, &zn1, CREATE_SUSPENDED, &ID0);
    hW[1] = (HANDLE)_beginthreadex( NULL, 0, ZNAK, &zn2, CREATE_SUSPENDED, &ID1);
    hW[2] = (HANDLE)_beginthreadex( NULL, 0, ZNAK, &zn3, CREATE_SUSPENDED, &ID2);

    SetThreadPriority(hW[0], THREAD_PRIORITY_LOWEST);
    SetThreadPriority(hW[1], THREAD_PRIORITY_**HIGHEST);
    SetThreadPriority(hW[2], THREAD_PRIORITY_LOWEST);

    for(i=0; i < MAX_THREADS; i++ ) ResumeThread(hW[i]);

    WatekGlowny(80, 50000, '*');

    for(i=0; i < MAX_THREADS; i++) CloseHandle(hW[i]);
    cout << "\n KONIC MAIN";
    // cin.get();
    return 0;
}

UINT WINAPI ZNAK(LPVOID lpzn)
{
    for(int i = 0; i < 60; i++) {
        for (int k=0; k<95000; k++) cos(sin(pow(log(k+1),2.2))); // opóźniacz
        cout << *((char*)lpzn) << ' ';
        cout.flush();
    }
}

void WatekGlowny(int n1, int n2, char zn) // długotrwałe obliczenia
{
    for (int k1=0; k1 < n1; k1++) {
        for (int k2=0; k2 < n2; k2++) pow(sin(k1),3.3)* pow(cos(k1),2.2);
        cout << zn;
    } cout << endl;
}
```

3.8. Funkcja CreateThread

Funkcja CreateThread jest funkcją systemu Windows, tworzącą nowy wątek.

HANDLE **CreateThread** (

```
❶ LPSECURITY_ATTRIBUTES lpAttributes,      // wskaźnik do struktury bezpieczeństwa
❷ DWORD dwStack,                // rozmiar początkowy stosu
❸ LPTHREAD_START_ROUTINE lpStartAddress,    // wskaźnik do funkcji wątku
❹ LPVOID lpParameter,          // argumenty dla wątku
❺ DWORD dwFlags,               // flagi
❻ DWORD lpThreadId             // zwraca wskaźnik identyfikatora wątku
);
```

Funkcja zwraca **uchwyt** wątku, zaś w razie niepowodzenia wartość **NULL**.

Wywołanie funkcji **CreateProcess** prowadzi do powstania głównego wątku procesu.

☛ Utworzenie pochodnych wątków wymaga wywołania funkcji **CreateThread** w już działającym wątku.

Wywołanie **CreateThread** tworzy w jądrze ObiektWątek jako strukturę danych, używaną przez SO do zarządzania wątkiem.

Nowy wątek działa w kontekście procesu, który go utworzył.

☛ Pamięć potrzebną na stos wątku, system alokuje w przestrzeni adresowej procesu.

lpAttributes -wskaźnik do struktury SECURITY_ATTRIBUTES.

Ustawiony na **NULL** powoduje użycie standardowych atrybutów bezpieczeństwa w ObiekcieWątku.

Podanie struktury SECURITY_ATTRIBUTES ze składową *binheritedHandle* ustawioną na TRUE pozwala każdemu potomnemu odziedziczyć uchwyt do nowego ObiektuWątku,

dwStack -wielkość przestrzeni adresowej [bajt] przeznaczonej na stos wątku.

Każdy wątek ma własny stos.

Wartość **0 (zero)** przydziela ten sam rozmiar jaki ma proces macierzysty.

Gdy funkcja **CreateProcess** tworzy proces, wywołuje w środku funkcję **CreateThread** aby zainicjalizować główny wątek procesu.

W miejscu parametru **dwStack** funkcja **CreateProcess** podaje wartość zapisaną w pliku wykonywalnym.

Można wpłynąć na tę wartość kluczem /STACK programu łączącego:

/STACK: [*zapas*] [*porcja*],

zapas określa ilość przestrzeni adresowej, którą system powinien zarezerwować dla stosu wątku (standardowo 1 MB);

porcja określa ilość fizycznej pamięci, jaką system powinien wstępnie przeznaczyć na stos (standardowo jedna strona).

lpStartAddress -adres funkcji wątkowej, od której rozpocznie się wykonanie nowego wątku

Prototyp funkcji:

```
DWORD WINAPI ThreadFunc( LPVOID );
```

lpParameter -jest przekazywany do funkcji wątkowej w momencie jej uruchomienia i informuje o danych inicjalizujących funkcję wątkową.

Może być wartością numeryczną, albo wskaźnikiem do struktury danych.

dwFlags -przekazuje dodatkowe flagi wpływające na sposób tworzenia wątku.

Dla wartości **0 (zero)** wątek po utworzeniu zarezerwowany jest do wykonania.

Dla wartości **CREATE_SUSPENDED** system tworzy i inicjalizuje wątek, ale wprowadza go w stan zawieszenia.

Umożliwia to aplikacji zmienić właściwości wątku przed jego uruchomieniem (np. priorytet).

☛ Zawieszony wątek wznowia się wywołując funkcję **ResumeThread**(uchwyt).

lpThreadId -adres zmiennej, w której funkcja **CreateThread** zapisuje identyfikator (ID) przypisany przez system nowemu wątkowi.

UWAGA. Pisząc kod w C++ nie należy wywoływać funkcji **CreateThread**.

Zaleca się używać funkcji z biblioteki czasu wykonywania np. **_beginthreadex**.¹

Funkcja **_beginthreadex** ma tę samą listę parametrów, co funkcja **CreateThread**, ale nazwy i typy tych parametrów nie pokrywają się idealnie.

Wynika to z przyjętego założenia, że funkcje biblioteki czasu wykonywania C++ powinny być niezależne od Windows'owych typów danych.

¹ Jeffrey Richter. Programowanie Aplikacji dla Microsoft Windows. RM, Warszawa 2002.

```
#include <iostream>
#include <windows.h>
#include <process.h>
#include <cmath>
using namespace std;

#define MAX_THREADS 3

DWORD WINAPI ZNAK(LPVOID);

void WatekGlowny(int, int, char);

int main() // Watek5a - priorytety
{
    int i;
    DWORD Th1, Th2, Th3;
    DWORD ID0, ID1, ID2;
    HANDLE hW[ MAX_THREADS ];
    char zn1 = '1', zn2 = '2', zn3 = '3';
    hW[0] = CreateThread( NULL,0, ZNAK, &zn1, CREATE_SUSPENDED, &ID0);
    hW[1] = CreateThread( NULL,0, ZNAK, &zn2, CREATE_SUSPENDED, &ID1);
    hW[2] = CreateThread( NULL,0, ZNAK, &zn3, CREATE_SUSPENDED, &ID2);
        SetThreadPriority(hW[0], THREAD_PRIORITY_LOWEST);
        SetThreadPriority(hW[1], THREAD_PRIORITY_HIGHEST);
        SetThreadPriority(hW[2], THREAD_PRIORITY_LOWEST)
    for(i=0; i < MAX_THREADS; i++ ) ResumeThread(hW[i]); // start wątków
        WatekGlowny(80, 50000, '*');
    for(i=0; i < MAX_THREADS; i++) CloseHandle(hW[i]);
    cout << "\n KONIEC MAIN";
    // cin.get();
    return 0;
}

DWORD WINAPI ZNAK(LPVOID lpzn)
{
    for(int i = 0; i < 60; i++) {
        for (int k=0; k<95000; k++) cos(sin(pow(log(k+1),2.2))); // spowalniacz
        cout << *((char*)lpzn) << ' ';
        cout.flush();
    }
}

void WatekGlowny(int n1, int n2, char zn) // długotrwałe obliczenia
{
    for (int k1=0; k1 < n1; k1++) {
        for (int k2=0; k2 < n2; k2++) pow(sin(k1),3.3)* pow(cos(k1),2.2);
        cout << zn;
    } cout << endl;
}
```

Zadanie 5.1

Napisać program wątkowy (**PG**), generujący 3 wektory (o zadanej wielkości) liczb typu double.

Wektor **A[10000]**: liczby z zakresu (1.1, 0.9)

Wektor **B[20000]**: liczby z zakresu (-1.1, -0.9)

Wektor **C[30000]**: liczby z zakresu (222, 888)

Dwie opcje programu:

1. Wątek (jeden) programu **PG** wyświetli kolejno fragmenty zawartości wektorów A, B, C.

2. Proces pochodny programu **PG** wyświetli kolejno fragmenty zawartości wektorów A, B, C.

1.11	8.22	4.18	7.67	7.35	1.23	2.4	5.02
1.18	6.13	2.56	4.2	7.99	3.76	9.51	2.35
8.69	8.52	6.48	3.72	4.41	1.59	3.53	7.18
-6.06	-6.25	-8.98	-2.63	-5.62	-1.9	-2.56	-2.15
-4.43	-6.4	-6.93	-1.6	-8.16	-8.81	-9.25	-5.17
-2.94	-9.87	-4.55	-8.49	-1.92	-1.18	-3.5	-8.47
350.74	541.6	770.01	794.06	424.46	464.73	880.36	225.11
576.09	626.37	522.23	626.72	568.24	705.97	581.21	378.73
741.25	887.8	399.3	237.81	673.02	833.91	613.55	705.84

Zadanie 5.2

Zmodyfikować program **Watek3a**, wykorzystując technikę nadawania priorytetów wątkom.

Celem modyfikacji jest zagwarantowanie pierwszeństwa generowania danych nad ich sortowaniem.

Zadanie 5.3

Zmodyfikować zadania z pliku **SO1LAB1** w taki sposób aby działały w strukturze wątkowej.