

### 3. PROCESY

Proces jest instancją (egzemplarzem) działającego programu, w skład którego wchodzi:

1. **ObiektJądra**, za pomocą którego system zarządza procesem i przechowuje statystyczne informacje o procesie.
2. Przestrzeń adresowa, zawiera kod i dane modułu wykonywalnego lub DLL, oraz pamięć alokowaną dynamicznie na stosy lub sterty wątku.

**Proces jest beczynny**. Musi uruchomić wątek w swoim kontekście, który wykona kod zawarty w przestrzeni adresowej procesu.

Gdyby zabraknie wątku wykonującego kod w przestrzeni adresowej procesu, **system** automatycznie **usuwa proces** z jego przestrzeni adresowej.

Jeden proces może zawierać wiele wątków wykonujących się.

Każdy wątek ma własny zbiór rejestrów CPU i własny stos.

SO przydziela każdemu wątkowi ok. 20 ms kwantu czasu CPU, stosując metodę cykliczną, co **sprawia wrażenie**, że wszystkie wątki wykonują się jednocześnie.

**W momencie uruchamiania Procesu system automatycznie tworzy jego pierwszy wątek, zwany głównym, który może tworzyć wątki dodatkowe (potomne).**

#### 3.1. Funkcja CreateProcess

BOOL **CreateProcess**(

- |    |                       |                           |   |
|----|-----------------------|---------------------------|---|
| 1  | LPCTSTR               | <b>lpApplicationName,</b> | // address of application name            |
| 2  | LPCTSTR               | <b>lpCommandLine,</b>     | // address of command line                |
| 3  | LPSECURITY_ATTRIBUTES | <b>lpProcess,</b>         | // address of process security attributes |
| 4  | LPSECURITY_ATTRIBUTES | <b>lpThread,</b>          | // address of thread security attributes  |
| 5  | BOOL                  | <b>bInheritHandles,</b>   | // new process <b>inherits handles</b>    |
| 6  | DWORD                 | <b>dwCreate,</b>          | // creation flags oraz priorytety procesu |
| 7  | LPOVOID               | <b>lpEnvironment,</b>     | // address of new environment block       |
| 8  | LPCTSTR               | <b>lpCurDir,</b>          | // address of current directory name      |
| 9  | LPSTARTUPINFO         | <b>lpStartInfo,</b>       | // address of STARTUPINFO                 |
| 10 | LPPROCESS_INFORMATION | <b>lpProcInfo</b>         | // address of PROCESS_INFORMATION         |

);

W przypadku powodzenia funkcja zwraca TRUE, inaczej FALSE (call GetLastError).

**lpApplicationName** -nazwa pliku wykonywalnego używanego przez nowy proces lub NULL.

➤ Należy podać rozszerzenie pliku (**nie** przyjmuje domyślnie **.exe**).

Funkcja zakłada, że jest to plik z bieżącego katalogu, chyba że podana będzie pełna ścieżka.

**Jeżeli funkcja nie znajdzie tego pliku, nie będzie przeglądać innych katalogów i zakończy się niepowodzeniem.**

**lpCommandLine** -napis z linią polecenia, którą funkcja używa do tworzenia procesu.

Może składać się z kilku słów tekstu, zwanych **leksemami**.

gdy **1-szy** parametr **lpApplicationName** = **NULL** to funkcja **CreateProcess** pobiera z **lpCommandLine** jego pierwszy **leksem** i **zakłada**, że jest to nazwa pliku wykonywalnego **.exe**.

Jeśli nie podano ścieżki, funkcja **CreateProcess** sprawdza kolejno katalogi:

1. zawierający plik **.exe** wywołującego procesu,
2. bieżący wywołującego procesu,
3. Windows,
4. wymienione w zmiennej środowiskowej PATH.

Procedura startowa C++ sprawdza linię polecenia procesu i przekazuje do funkcji **WinMain** (main) przez parametr **lpCmdLine** adres pierwszego argumentu po nazwie pliku wykonywalnego.

**lpProcess, lpThread** -ustawia atrybuty bezpieczeństwa nowo powstałych obiektów jądra ObiektProces i ObiektWątek.

➤ Wartość **NULL** ustawia standardowe deskryptory zabezpieczeń.

Można utworzyć struktury SECURITY\_ATTRIBUTES, i na ich bazie zbudować własny system zabezpieczeń. Są one też pomocne przy przekazywaniu metodą dziedziczenia uchwytów któregoś z tych obiektów do procesu potomnego.

**bInheritHandles** -współpracuje ze strukturami SECURITY\_ATTRIBUTES w celu realizowania dziedziczenia uchwytów.

Przyjmuje wartość **TRUE** (dziedziczenie) lub **FALSE**.

**dwCreate** -flagi wpływające na sposób tworzenia procesu.

Operator sumy bitowej OR, umożliwia tworzenie kombinacji kilku flag jednocześnie.

DEBUG\_PROCESS informuje system, że proces nadrzędny chce debugować procesy potomne, bez powiadamiania procesu debugującego o wystąpieniu pewnych zdarzeń w procesach debugowanych.

DEBUG\_ONLY\_THIS\_PROCESS proces debugujący jest powiadamiany wyłącznie o specjalnych zdarzeniach i to tylko w swoim bezpośrednim procesie potomnym.

**CREATE\_SUSPENDED** zawiesza wątek główny procesu, zanim proces wykona jakiegokolwiek kod. Wznowienie działania procesu potomnego realizuje funkcja **Resume\_Thread**.

DETACHED\_PROCESS blokuje procesowi opartemu na CUI dostęp do okna konsolowego rodzica. Powoduje skierowanie strumienia wyjściowego procesu potomnego do nowego okna konsolowego.

**CREATE\_NEW\_CONSOLE** każe utworzyć nowe okno konsolowe dla nowego procesu.

Jednocześnie użycie CREATE\_NEW\_CONSOLE i DETACHED\_PROCESS daje błąd.

CREATE\_NO\_WINDOW blokuje tworzenie przez system okien konsolowych dla aplikacji.

CREATE\_NEW\_PROCESS\_GROUP użyta podczas tworzenia nowego procesu CUI utworzy nową grupę procesów.

Naciśnięcie klawiszy [Ctrl+C] lub [Ctrl+Break], gdy jest aktywny któryś z procesów tej grupy, spowoduje powiadomienie o tym tylko procesów z tej grupy.

CREATE\_DEFAULT\_ERROR\_MODE proces potomny nie będzie dziedziczyć trybu błędowego używanego przez proces nadrzędny.

CREATE\_SEPARATE\_WOW\_VDM każe systemowi utworzyć oddzielną Wirtualną Maszynę DOSową (Virtual DOS Machine - VDM) i uruchomić w niej aplikację 16-bitowej wersji Windows.

CREATE\_SHARED\_WOW\_VDM umożliwia uruchamianie wielu aplikacji 16-bitowej wersji Windows we wspólnej maszynie VDM systemu.

CREATE\_UNICODE\_ENVIRONMENT informuje system, że blok środowiskowy procesu potomnego powinien zawierać znaki Unicode. Standardowo blok ten zawiera napisy ANSI.

CREATE\_FORCEDOS zmusza system do uruchomienia aplikacji DOSowej osadzonej wewnątrz aplikacji 16-bitowej wersji OS/2.

CREATE\_BREAKAWAY\_FROM\_JOB pozwala procesowi należącemu do **zadania** utworzyć nowy proces poza tym zadaniem.

**lpEnvironment** -przekazuje blok pamięci z napisami środowiskowymi, których ma używać nowy proces. Z reguły przyjmuje wartość NULL, co powoduje dziedziczenie przez proces potomny wszystkich zmiennych środowiskowych, do których ma dostęp jego rodzic.

**lpCurDir** -pozwala procesowi nadrzédnemu ustawić bieżący katalog i napęd procesu potomnego. Jeśli parametr ten ma wartość **NULL**, katalog roboczy nowego procesu będzie taki sam jak aplikacji uruchamiającej ten proces.

W przeciwnym razie parametr **pszCurDir** musi wskazywać łańcuch z żądanym napędem i katalogiem roboczym. Podawana nazwa ścieżki musi zawierać literę napędu.

**lpStartInfo** - wskazuje strukturę STARTUPINFO, której składowe używane są przez system w momencie tworzenia nowego procesu.

Większość aplikacji uruchamia proces potomny, stosując domyślne wartości.

#### ► Minimum działań przy tworzeniu procesu potomnego:

- zainicjalizować składowe struktury STARTUPINFO wartością **0**
- podać w składowej **cb** rozmiar całej struktury STARTUPINFO .
- utworzyć zmienną strukturalną **pi** .

```
STARTUPINFO si = {0};  
si.cb = sizeof(STARTUPINFO);  
PROCESS_INFORMATION pi;
```

- ➔ Funkcja **CreateProcess** zwraca TRUE **przed** zakończeniem inicjalizacji procesu, zanim nastąpi lokalizacja potrzebnych bibliotek DLL.

Jeśli któreś z nich nie uda się poprawnie zainicjalizować, nastąpi zamknięcie procesu i proces nadrzędny nie dowie się o tym zdarzeniu.

Główny wątek zaczyna swoje działanie od wykonania kodu startowego czasu wykonywania C++, a ten z kolei wywołuje funkcję startową aplikacji **WinMain** lub **main**.

Po wywołaniu **CreateProcess**, system operacyjny tworzy:

- w jądrze **ObiektProces** z licznikiem użyć ustawionym na 1; jest to strukturą używa do zarządzania procesem, przechowywanie informacji statystycznych o procesie.
- w jądrze **ObiektWątek** (z licznikiem użyć ustawionym na 1), odpowiadający głównemu wątkowi nowego procesu; jest to strukturą danych używaną do zarządzania właściwym wątkiem.
- wirtualną **przestrzeń adresową** nowego procesu i ładuje do niej kod oraz dane z pliku wykonywalnego, a także potrzebne moduły DLL.

```
typedef struct _STARTUPINFO {           // si  
    ❶ DWORD    cb;                      // należy ustawić na wartość sizeof(STARTUPINFO)  
    ❷ LPTSTR    lpReserved,             // NULL  
    ❸ LPTSTR    lpDesktop,              // NULL ustawia proces na bieżący pulpit  
    ❹ LPTSTR    lpTitle,                // tytuł okna konsolowego, jeżeli NULL to nazwa pliku wykonywalnego  
    ❺ DWORD     dwX,                    // współrzędne x okna procesu potomnego na ekranie  
    ❻ DWORD     dwY,                    // współrzędne y okna procesu potomnego na ekranie  
    ❼ DWORD     dwXSize,                // szerokość [piksel] okna aplikacji  
    ❽ DWORD     dwYSize,                // wysokość [piksel] okna aplikacji  
    ❾ DWORD     dwXCountChars,          // szerokość [znak] okna konsolowego procesu potomnego  
    DWORD     dwYCountChars,          // wysokość [znak] okna konsolowego procesu potomnego  
    DWORD     dwFillAttribute,         // kolor tekstu i tła okna konsoli procesu potomnego  
    DWORD     dwFlags,                 // określają, które parametry będą wykorzystane (tabela niżej)
```

WORD	wShowWindow,	// sposób wyświetlania 1-go okna procesu potomnego
WORD	cbReserved2,	// wartość zero (0)
LPBYTE	lpReserved2,	// NULL
HANDLE	hStdInput,	// uchwyt bufora na konsolowy strumień Wejścia (stand. klawiatura)
HANDLE	hStdOutput,	// uchwyt bufora na konsolowy strumień Wyjścia (stand. okno konsoli)
HANDLE	hStdError,	// uchwyt bufora na konsolowy strumień Wyjścia (stand. okno konsoli)

```
} STARTUPINFO, *LPSTARTUPINFO;
```

STARTF_USEPOSITION	użyć składowych <i>dwX</i> i <i>dwY</i> .
<b>STARTF_USESIZE</b>	użyć składowych <b><i>dwXSize</i></b> i <b><i>dwYSize</i></b> .
STARTF_USECOUNTCHARS	użyć składowych <i>dwXCountChars</i> i <i>dwYCountChars</i>
STARTF_USESHOWWINDOW	użyć składowej <i>wShowWindow</i> .
STARTF_USEFILLATTRIBUTE	użyć składowej <i>dwFillAttribute</i> .
STARTF_USESTDHANDLES	użyć składowych <i>hStdInput</i> , <i>hStdOutput</i> i <i>hStdError</i> .
STARTF_RUN_FULLSCREEN	tryb pełnoekranowy dla aplikacji konsolowej na komputerze x86

**lpProcInfo** -wskazuje strukturę PROCESS\_INFORMATION, którą trzeba zaalokować przed wywołaniem funkcji **CreateProcess**.

Jako minimum wystarczy wykonać:

```
PROCESS_INFORMATION pi;  
CreateProcess(..., &si, &pi);
```

```
typedef struct _PROCESS_INFORMATION { // pi  
    HANDLE hProcess;                  // uchwyt procesu  
    HANDLE hThread;                  // uchwyt wątku  
    DWORD dwProcessID;               // identyfikator procesu  
    DWORD dwThreadID;                // identyfikator wątku  
} PROCESS_INFORMATION;
```

➔ Uruchomienie nowego procesu tworzy 2 obiekty jądra:

**ObiektProces** oraz **ObiektWątek**.

Na początku system ustawia licznik użyć każdego z tych Obiektów na **1**.

Następnie **CreateProcess** otwiera **ObiektProces** oraz **ObiektWątek** i umieszcza ich uchwyt (względem procesu) w składowych **hProcess** i **hThread**.

Po otwarciu tych obiektów, wewnątrz funkcji *CreateProcess* ich liczniki użyć przyjmują wartość **2**.

Aby zwolnić **ObiektProces**, trzeba zakończyć odpowiadający mu proces (i zmniejszyć licznik użyć o 1) oraz zamknąć uchwyt **ObiektProces** w procesie nadrzédnym, wywołując funkcję *CloseHandle* (i ponownie zmniejszyć licznik użyć o 1, czyli do zera).

Aby zwolnić **ObiektWątek**, trzeba zakończyć sam wątek i zamknąć jego uchwyt w procesie nadrzédnym.

Dla nowego **ObiektProces** i **ObiektWątek** system przypisuje unikatowe identyfikatory **ID**.

- ➔ Przed powrotem funkcja **CreateProcess** wstawia ID procesu i ID wątku do składowych *dwProcessID* i *dwThreadID* .

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

int main( int argc, char* argv[ ] )           // Proces1
{
    char cmd1[ ] = "Prog1 ";                 // nazwa skompilowanego programu
    char cmd2[ ] = "Prog1 24 225 81";
    char cmd3[ ] = "explorer";

    STARTUPINFO si = {0};
    si.cb = sizeof(STARTUPINFO);
    PROCESS_INFORMATION pi;

    CreateProcess(0, cmd1, 0,0,0,0,0, &si, &pi); // process potomny
    CreateProcess(0, cmd2, 0,0,0,0,0, &si, &pi);
    BOOL OK = CreateProcess(0, cmd3, 0,0,0,0,0, &si, &pi);
    if(!OK) { cout << "CreateProcess error: " << GetLastError(); cin.get(); return 1; }

    // -----wypisz parametry programu Proces1
    cout << "Parametry wywołania programu:" << endl;
    for(int i = 0; i < argc; i++) cout << "argv[" << i << "] -> " << argv[i] << endl;
    cout << "Koniec MAIN \n";
    //cin.get();
    return 0;
}
```

Skompilować program **Prog1** (z części 1-szej).  
Program **Proces1** uruchomić z **wiersza poleceń**  
Należy uruchamiać program **Proces1** wielokrotnie i obserwować strukturę wyników.  
Czy za każdym razem jest identyczna jak podana obok?  
Jeżeli inna przy każdym uruchomieniu, to dlaczego?

E:\SO\_LAB>Proces1 987 765 432

**Brak parametrów w Prog1**

Koniec Prog1  
sqrt(24) = 4.89898  
sqrt(225) = 15  
sqrt(81) = 9  
Koniec Prog1  
Parametry wywołania programu:  
argv[0] -> Proces1  
argv[1] -> 987  
argv[2] -> 765  
argv[3] -> 432  
Koniec MAIN

Uwaga: przed uruchomieniem **CreateP1** utworzyć na dysku dwa pliki tekstowe.

```
// CreateP1 – 1-szy i 2-gi parametr

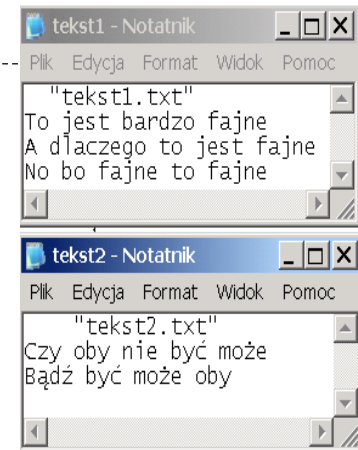
#include <windows.h>
int WINAPI WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int iCmdShow )
{
    BOOL ok1, ok2;
    char program[ ] = "c:\\Windows\\notepad.exe";
    char linia1[ ] = "E:\\tekst1.txt";
    char linia2[ ] = "notepad E:\\tekst2.txt";

    STARTUPINFO si = {0};
    si.cb = sizeof(STARTUPINFO);
    PROCESS_INFORMATION pi;

    ok1 = CreateProcess(program, linia1, 0, 0, 0, 0, 0, 0, &si, &pi);
    ok2 = CreateProcess( 0, linia2, 0, 0, 0, 0, 0, 0, &si, &pi);

    HANDLE uchwytyProcesu2 = pi.hProcess;

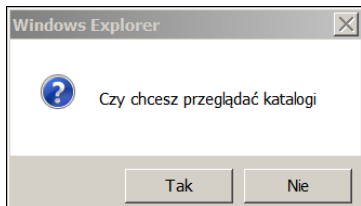
    return 0;
}
```



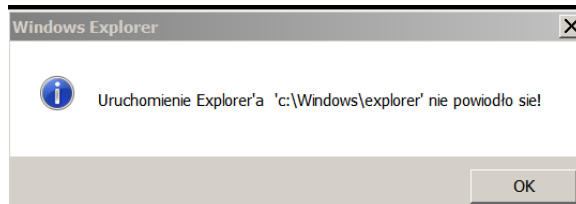
```
#include <windows.h>
#include <stdio.h>
using namespace std;

int main() // CreateP2
{
    char nazwaOkna[] = "Windows Explorer";
    char komunikat[] = "Czy chcesz przeglądać katalogi";
    char program[] = "c:\\Windows\\explorer";
    char bufor[128];
    int ok = MessageBoxEx(NULL, komunikat, nazwaOkna, MB_YESNO | MB_ICONQUESTION, 0);
    if (ok == IDYES) {
        STARTUPINFO si = {0};
        si.cb = sizeof(STARTUPINFO);
        PROCESS_INFORMATION pi;
        BOOL ok1 = CreateProcess(program, 0, 0, 0, 0, 0, 0, 0, &si, &pi);
        if (!ok1) {
            wsprintf(bufor, "Uruchomienie Explorer'a '%s' nie powiodło się!\0", program);
            MessageBoxEx(NULL, bufor, nazwaOkna, MB_OK | MB_ICONINFORMATION, 0);
        }
        HANDLE uchwytyProcesu = pi.hProcess;
        printf("uchwytyProcesu = %p", uchwytyProcesu);
    }
    // getchar();
    return 0;
}
```

uchwytyProcesu = 0000006C



Po wciśnięciu przycisku **Tak** pojawia się komunikat:



### Uwaga:

Poprawić program **CreateP2** aby zlikwidować niepowodzenie uruchomienia **eksploraer'a**.

### □ Program tradycyjny i **wieloprocesowy** – **kiedy to ma sens?**

```
#include <stdio.h>
const int COL = 5000;
int main(int argc, char **argv)
{
    FILE *OUT;
    int i, k, row = 10;
    double A = new double[row][COL]

    GenMac(row, COL, A);
    SaveMac(OUT, row, COL, A);
    BubbleSort(A, row, COL);

    OBLICZENIA_PROCESU_main

    return 0;
}

void GenMac( )
{
    // Generuje Tablicę M
}
void SaveMac( )
{
    // Zapisuje tablicę M do pliku
}
void BubbleSort( )
{
    // sortuje tablicę M
}
```

```
#include <stdio.h>
const int COL = 5000;
int main(int argc, char **argv)
{
    int i, row = 10;
    double A = new double[row][COL]
    CreatePocess("GenMac.exe", ... );
    CreatePocess("SaveMac.exe", ... );
    CreatePocess("BubbleSort.exe", ... );
    OBLICZENIA_PROCESU_main

    return 0;
}
```

Program **CreateP3** uruchamia **dwa** procesy potomne w osobnych oknach, których parametry ustawiane są w **9-tym** parametrze funkcji **CreateProcess**.

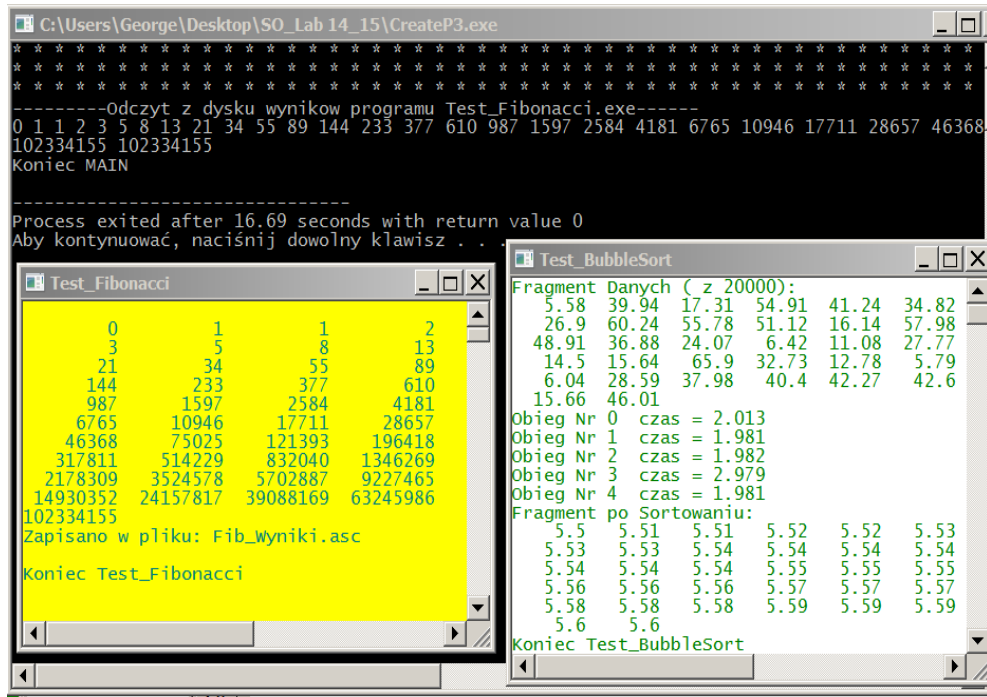
**Test\_BubbleSort.exe** Generuje kilka serii liczb losowych i Sortuje je.

**Test\_Fibonacci.exe** Generuje rekurencyjnie ciąg liczb i **zapisuje** je do zbioru dyskowego.

Parametry dla obu procesów przekazywane są z programu **CreateP3** (PROCES MACIERZYSTY).

Funkcja **ProgramGlowny** jest funkcją pomocniczą, symulującą długie obliczenia numeryczne.

$$\text{Fib}(n) = \begin{cases} n & \text{dla } n < 2 \\ \text{Fib}(n-2) + \text{Fib}(n-1) & \text{dla } n \geq 2 \end{cases}$$



```
#include<windows.h>
#include <stdio>
#include <stdlib>
#include <cmath>
using namespace std;
void ProgramGlowny(int, char);

int main(int argc, char *argv[ ]) // CreateP3
{
    PROCESS_INFORMATION pi1, pi2;
    char cmd1[80] = "Test_BubbleSort 25000 5.5 66.6";
    STARTUPINFO si1 = {
        sizeof(si1), NULL, NULL, "Test_BubbleSort", 500, 370,
        420, 310, // szerokość i wysokość okna aplikacji [piksel]
        0, 0,
        0x1d, // atrybuty wypełnienia okna
        STARTF_USEFILLATTRIBUTE | STARTF_USEPOSITION | STARTF_USESIZE,
        0, 0, NULL, NULL, NULL
    };
    char cmd2[80] = "Test_Fibonacci 40 D:\\Fib.asc";
    STARTUPINFO si2 = {
        sizeof(si2), NULL, NULL, "Test_Fibonacci", 500, 10, 420, 310, 0, 0,
        0x2e, // atrybuty wypełnienia okna
        STARTF_USEFILLATTRIBUTE | STARTF_USEPOSITION | STARTF_USESIZE,
        0, 0, NULL, NULL, NULL
    };
    CreateProcess(0, cmd1, NULL, NULL, 0, CREATE_NEW_CONSOLE, NULL, NULL, &si1, &pi1);
    CreateProcess(0, cmd2, NULL, NULL, 0, CREATE_NEW_CONSOLE, NULL, NULL, &si2, &pi2);
    ProgramGlowny(260, '*'); // 1-szy parametr decyduje o powodzeniu poniższego odczytu ?

    long F; // -----odczyt kontrolny wyników programu Test_Fibonacci.exe
    FILE *pF = fopen("Fib_Wyniki.asc", "rt");
    while (!feof(pF)) { fscanf(pF, "%ld", &F); printf("%ld ", F); } fclose(pF);

    puts("\nKoniec MAIN");
    // getchar();
    return 0;
}

void ProgramGlowny(int n, char zn)
{
    for (int k1=0; k1 < n; k1++) {
        for (int k2=0; k2 < 90000; k2++) pow(sin(k1),3.3)* pow(cos(k1), 2.2);
        printf("%c ", zn);
    } puts("");
}
```

**Zadanie 3.1.** Odczyt kontrolny zawartości pliku funkcjami języka C++, zastąpić funkcjami Systemu Operacyjnego.



### 3.2. Procesy potomne

Tradycyjny program składa się z wielu funkcji, które często korzystają z *globalnych struktur danych* do wymiany informacji między sobą.

Może dojść do naruszenia przestrzeni adresowej procesu przez fragmenty kodu zawarte w jego funkcjach. Aby zabezpieczyć się przed taką ewentualnością można utworzyć z **funkcji** oddzielne **procesy** (zwane potomnymi).

→ W czasie działania procesu potomnego można zawiesić wykonanie głównego kodu programu (aż do zakończenia potomnego) lub kontynuować je równolegle.

Proces potomny może pracować na danych zawartych w przestrzeni adresowej głównego procesu.

Proces potomny można uruchomić w jego własnej przestrzeni adresowej i przyznać prawa dostępu tylko do wybranych danych w przestrzeni procesu nadrzędnego.

Windows oferuje kilka metod transferu danych między różnymi procesami:

DDE (Dynamic Data Exchange),  
łącza, gniazda pocztowe.

→ Wygodną techniką współużytkowania danych są **pliki mapowane** w pamięci.

```
// szablon tworzy proces potomny, zaś proces nadrzędny czeka na jego zakończenie
PROCESS_INFORMATION pi;
DWORD dwExitCode;
BOOL OK = CreateProcess(0, "Potomny", ..., &pi); // utworzenie procesu potomnego
if (OK) {
    CloseHandle(pi.hThread); // zamknięcie uchwytu wątku, gdyż nie jest już potrzebny
    WaitForSingleObject(pi.hProcess, INFINITE); // zawieszenie wykonania do momentu zakończenia potomka
    GetExitCodeProcess(pi.hProcess, &dwExitCode); // proces potomny zakończony; pobranie kodu wyjścia
    CloseHandle(pi.hProcess); // zamknięcie uchwytu procesu, gdyż nie jest już potrzebny
}
```

**WaitForSingleObject** czeka, aż Obiekt wskazywany przez 1-szy parametr zakończy działanie.

Wywołanie **WaitForSingleObject** zawiesza wątek procesu nadrzędnego do momentu zakończenia procesu potomnego.

Kod wyjścia procesu potomnego udostępnia funkcja **GetExitCodeProcess**.

**Uwaga: funkcja WaitFor.... omówiona jest w pliku SO\_LAB6**

Funkcja **CloseHandle** zmniejsza licznik użyć Obiektu **Procesu** i Obiektu **Wątku** do 0, co pozwala zwolnić zajmowaną przez nie pamięć.

→ Zamknięcie uchwytu głównego wątku procesu potomnego zaraz po powrocie z funkcji **CreateProcess** **nie** powoduje zakończenia głównego wątku potomka, jedynie zmniejsza o 1 licznik użyć jego Obiektu ← **działanie prawidłowe**.

Niech główny wątek procesu potomnego tworzy następny wątek i zaraz główny kończy swoje działanie.

W tym momencie system może zwolnić z pamięci Obiekt Wątek głównego wątku potomka, o ile tylko proces nadrzędny nie ma ważnego uchwytu do tego Obiektu.

W przeciwnym razie system musi zaczekać ze **zwolnieniem pamięci** aż do momentu zamknięcia uchwytu przez proces nadrzędny.

### □ Niezależne procesy potomne

Aplikacja może uruchamiać inne procesy jako **procesy niezależne**.

Oznacza to, że po uruchomieniu procesu potomnego jego **rodzic**:

- nie musi komunikować się z nowym procesem,
- nie oczekuje na jego zakończenie,
- dalej kontynuuje swoje działanie.

→ Zerwanie związków z procesem **potomnym** wymaga, aby:

Proces nadrzędny zamknął uchwyt **nowego** procesu i jego **głównego** wątku za pomocą funkcji **CloseHandle**

```
// Uruchomienie całkowicie niezależnego nowego procesu
PROCESS_INFORMATION pi;
BOOL OK = CreateProcess(..., &pi); // proces potomny
if (OK) {
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
}
```

### 3.3. Zakończenie procesu

Sposoby zakończenia procesu:

1. Powrót z funkcji będącej punktem wejścia do głównego wątku (zalecany).
2. Naturalne zakończenie działania wszystkich wątków procesu.
3. Wywołanie w jednym z wątków procesu funkcji **ExitProcess** (zaleca się unikać).
4. Wywołanie w wątku innego procesu funkcji **TerminateProcess** (zaleca się unikać).

### □ Powrót z funkcji stanowiącej punkt wejścia wątku głównego

Proces aplikacji powinien kończyć się w momencie powrotu z funkcji stanowiącej punkt wejścia głównego wątku tej aplikacji.

Gwarantuje to zwolnienie wszystkich zasobów wykorzystywanych przez wątek główny.

- Obiekty C++ utworzone przez ten wątek zostaną usunięte za pomocą ich destruktorów.
- System operacyjny zwolni pamięć używaną przez stos wątku.
- System ustawi wartość powrotną funkcji będącej punktem wejścia głównego wątku na kod wyjścia z procesu (przechowywany w jądrze w Obiekcie **Procesie**).
- System zmniejszy licznik użyć Obiektu **Procesu** w jądrze.

### □ Wszystkie wątki procesu zakończyły się naturalnie

Gdy następuje zakończenie wszystkich wątków procesu (same wywołały funkcję **ExitThread** lub wywołano **TerminateThread**), system operacyjny uznaje, że należy **zlikwidować przestrzeń adresową procesu**, tym samym zakończyć sam proces.

- Kod wyjścia procesu ustawiany jest na kod wyjścia jego ostatnio zakończonego wątku.

## □ Funkcja `ExitProcess`

Wyjście z procesu następuje, gdy jeden z jego wątków wywoła funkcję **`ExitProcess`**:

`VOID ExitProcess(UINT uExitCode)`

`uExitCode` – wartość, którą funkcja zwróci do systemu

Funkcja kończy działanie wątków pomocniczych procesu i zwalnia załadowane biblioteki DLL.

Kod umieszczony za wywołaniem **`ExitProcess`** nigdy się nie będzie wykonywał.

Po wyjściu z funkcji będącej punktem wejścia głównego wątku (WinMain, main) następuje powrót do kodu startowego runtime C/C++, który jawnie wywołuje funkcję **`ExitProcess`** i przekazuje do niej wartość uzyskaną od funkcji będącej punktem wejścia twojej aplikacji.

Uwaga: Wywołanie **`ExitProcess`** lub **`ExitThread`** powoduje zakończenie procesu lub wątku w trakcie wykonywania kodu tej funkcji.

System operacyjny poprawnie zwolni wszystkie zasoby systemowe procesu lub wątku.

\* Aplikacja C/C++ powinna jednak unikać wywoływania tych funkcji, gdyż kod czasu wykonywania C/C++ może nie zdążyć posprzątać wszystkiego.

## □ Funkcja `TerminateProcess`

Funkcji należy używać wtedy, gdy nie można zmusić procesu do wyjścia za pomocą innych metod.

`BOOL TerminateProcess(HANDLE hProcess, UINT fuExitCode)`

`hProcess` – uchwyt procesu, który ma zostać zakończony

`fuExitCode` – wartość, będąca kodem wyjścia zamykanego procesu.

Wywołując **`TerminateProcess`**, wątek może zakończyć swój **własny** proces albo jakiś **inny**.

Zamykany proces nie otrzymuje powiadomienia o tym fakcie - **aplikacja nie może prawidłowo posprzątać po sobie**.

Mimo to po jego zakończeniu wszystko zostanie sprzątnięte przez **system operacyjny**, a jego zasoby zostaną odzyskane (zwolnienie całej pamięci wykorzystywanej przez proces, zamknięcie otwartych przez niego plików, zmniejszenie liczników użyć obiektów jądra).

**Zakończenie procesu nie doprowadzi do wycieku zasobów.**

**UWAGA:** **`TerminateProcess`** jest funkcją asynchroniczną - przekazuje polecenie zakończenia procesu, a następnie wraca, nie czekając na wynik działania.

➔ **Wywołanie jej nie daje gwarancji, że dany proces już nie działa.**

Aby się upewnić, można wywołać funkcję **`WaitForSingleObject`** lub podobną, przekazując do niej uchwyt zakończonego procesu.

**Gdy proces kończy się** wykonywane są następujące czynności:

1. zamknięcie wszystkich wątków procesu.
2. zwolnienie wszystkich obiektów USER i GDI alokowanych przez proces, a także zamknięcie wszystkich jego obiektów jądra.

*Obiekty jądra są usuwane tylko wtedy, gdy ich uchwyty nie są już otwarte w żadnym innym procesie. W przeciwnym wypadku nadal będą istniały.*

3. zmiana kodu wyjścia ze STILL\_ACTIVE na kod przekazany przez **`ExitProcess`**.
4. zasygnalizowanie ObiektuProcesu w jądrze.

Pozwala to wznowić wykonanie wątków zawieszonych w systemie z powodu oczekiwania na zakończenie procesu.

5. zmniejszenie o 1 licznika użyć ObiektuProcesu w jądrze.

**Obiekty jądra utworzone przez proces istnieją tak długo jak proces.**

**ObiektProcess może istnieć nawet dłużej.**

Gdy proces kończy działanie, system automatycznie **zmniejsza** licznik użyć odpowiadającego mu obiektu jądra.

Gdy licznik osiągnie **zero**, oznacza to, że żaden inny proces nie ma otwartego uchwytu tego ObiektuProcesu i można go zniszczyć.

Jeśli licznik użyć ObiektProcesu ma nadal wartość większą od zera, wskazuje to na istnienie otwartego jego uchwytu w jakimś innym procesie.

Zazwyczaj dzieje się tak w przypadku nie zamknięcia uchwytu procesu w jego procesie nadrzędnym. **Nie jest to błąd.**

**ObiektProces** przechowuje informacje statystyczne o procesie, przydatne po jego zakończeniu.

Proces nadrzędny może uzyskać informację, ile czasu CPU zużył jego potomek.

Można uzyskać kod wyjścia nie działającego już procesu za pomocą funkcji **`GetExitCodeProcess`**:

`BOOL GetExitCodeProcess(HANDLE hProcess, PDWORD pdwExitCode)`

`hProcess` – identyfikator ObiektuProcesu z którego struktury funkcja pobiera składową zawierającą kod wyjścia procesu a następnie udostępnia poprzez parametr **`pdwExitCode`**.

Funkcja **`GetExitCodeProcess`** może być wywoływana w dowolnym momencie.

Jeśli proces wykonuje się jeszcze, funkcja przekazuje w **`pdwExitCode`** identyfikator STILL\_ACTIVE (0x103), w przeciwnym faktyczny kod wyjścia procesu.

➔ Zakończenie procesu można wykryć cyklicznie wywołując funkcję **`GetExitCodeProcess`** i badając przekazany przez nią kod wyjścia tego procesu - **rozwiązanie nieefektywne**.

Jeżeli nie potrzebne są informacje statystyczne o procesie, należy poinformować o tym system, wywołując funkcję **`CloseHandle`**.

Jeśli proces ten przestał działać, funkcja **`CloseHandle`** zmniejszy licznik użyć jego ObiektuJądra i zwolni go.

## Zadanie 3.2

Zmodyfikować: **Zadanie 1.1, Zadanie 1.1a, Zadanie 1.2a, Zadanie 1.2a** z pliku **SO1\_LAB1** w taki sposób, aby część zadania wykonywał program główny, zaś pozostałą część procesy potomne.

## ANEKS 3.1. Linia polecenia procesu

- W momencie tworzenia nowego procesu system przekazuje do niego linię polecenia. Linia ta może zawierać **leksem** z nazwą pliku wykonywalnego użytego do uruchomienia procesu; może zawierać samo zero stojące na końcu łańcucha znaków.

Rozpoczęcie wykonania kodu startowego czasu wykonywania C/C++, pobiera linię polecenia **procesu**, **przeskakuje** w niej nazwę pliku wykonywalnego i przekazuje **wskaźnik do pozostałej** części linii przez parametr **lpCmdLine** funkcji **WinMain()**.

Parametr **pszCmdLine** funkcji **WinMain** wskazuje zawsze napis ANSI.

W funkcji **WinMain** można korzystać z Unicojowej wersji linii polecenia procesu.

Interpretacja linii polecenia w aplikacji może odbywać się w dowolny sposób.

- Można pisać bezpośrednio w buforze pamięci wskazywanym przez parametr **lpCmdLine**.

Zaleca się jednak traktować ten bufor jako **tylko** do czytania.

Chcąc coś zmienić w linii polecenia, powinno się najpierw przekopiować ją do lokalnego bufora w swojej aplikacji i dopiero wtedy rozpocząć modyfikację.

- Poprzez wskaźnik do całej linii polecenia procesu wywołując funkcję **GetCommandLine**:

PTSTR **GetCommandLine()**

Funkcja ta zwraca wskaźnik do bufora zawierającego kompletną linię polecenia, razem z pełną nazwą ścieżkową uruchomionego pliku.

- Linia** polecenia może składać się z wielu **leksemów**, będących parametrami procesu.

Dostęp do składników linii zapewniają zmienne **globalne**:

**\_\_argc** oraz **\_\_argv**.

extern int **\_\_argc**;

extern char **\*\*\_\_argv**;

**\_\_argc** has the value of argc passed to main when the program starts.

**\_\_argv** points to an array containing the elements of argv[ ] passed to main when the program starts.

```
void func()
{
    cout << "argc= " << __argc << endl;
    for (int i = 0; i < __argc; ++i) cout << __argv[i] << endl;
}

int main(int argc, char ** argv)
{
    func();
    // THIS FUNCTION KNOWS ALL THE main() ARGUMENTS
    return 0;
}
```

W wersji Unicode istnieje funkcja **CommandLineToArgvW**, która dzieli na leksemy napis Unicode.

PWSTR **CommandLineToArgvW** (PWSTR **pszCmdLine**, int\* **pNumArgs**);

**pszCmdLine** -wskazuje napis z całą linią polecenia,

**pNumArgs** -adres liczby całkowitej, określającej liczbę argumentów w linii.

Funkcja **CommandLineToArgvW** zwraca adres tablicy wskaźników do napisów Unicode.

Funkcja alokuje dodatkową pamięć.

Większość aplikacji nie zwalnia tej pamięci - zakładając, że zrobi to system operacyjny po zakończeniu całego procesu.

Chcąc samodzielnie zwolnić tę pamięć, należy wywołać funkcję **HeapFree**.

```
int nArgs;
PWSTR *pArgv = CommandLineToArgvW(GetCommandLineW(), &nArgs);
if (*pArgv[1] == 'pierwszy') // korzystanie z parametrów
{
    ...
}
HeapFree(GetProcessHeap(), 0, pArgv) // zwolnienie bloku pamięci
```

```
#include <windows.h> // Prog1Win, zmienne globalne __argc oraz __argv
#include <cstdlib>
#include <iostream>
#include <cmath>
using namespace std;
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int iCmdShow)
{
    double x;
    int i;
    cout << "Linia polecenia: " << lpCmdLine << endl;
    cout << "-----Program Prog1Win-----\n";
    cout << "argc= " << __argc << endl;
    for (i=0; i < __argc; i++) cout << __argv[i] << endl;
    if (__argc <= 1) cout << "brak parametrów ???\n";
    else
        for (i=1; i < __argc; i++) {
            x = atof(__argv[i]);
            cout << "sqrt(" << x << ") = " << sqrt(x) << endl;
        }
    cout << "Koniec Prog1Win";
    // cin.get();
    return 0;
}
```

```
D:\>Prog1win 6568 121
Linia polecenia: 6568 121
-----Program Prog1Win-----
argc= 3
D:\Prog1win
6568
121
sqrt(6568) = 81.0432
sqrt(121) = 11
Koniec Prog1Win
```



W programie **Create1\_Prog1Win** parametry dla procesu potomnego przekazywane są z wnętrza procesu macierzystego.

```
#include <windows.h> // Create1_Prog1Win

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int iCmdShow)
{
    char linia[ ] = "Prog1Win 625 5457 81";
    BOOL ok;
    STARTUPINFO si = {0};
    si.cb = sizeof(STARTUPINFO);
    PROCESS_INFORMATION pi;
    ok = CreateProcess(0, linia, 0, 0, 0, 0, 0, 0, &si, &pi);
    return 0;
}
```

```
Linia polecenia: 625 5457 81
-----Program Prog1Win-----
argc= 4
D:\Prog1Win.exe
625
5457
81
sqrt(625) = 25
sqrt(5457) = 73.8715
sqrt(81) = 9
Koniec Prog1Win
```

Program **Create2\_Prog1Win** przekazuje parametry dla procesu potomnego poprzez linię poleceń.

```
#include <windows.h> // Create2_Prog1Win
#include <stdio.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine, int iCmdShow)
{
    char program[ ] = "D:\\WinProg1.exe";
    BOOL ok;
    STARTUPINFO si = {0};
    si.cb = sizeof(STARTUPINFO);
    PROCESS_INFORMATION pi;
    ok = CreateProcess(program, lpCmdLine, 0, 0, 0, 0, 0, 0, &si, &pi);
    if (!ok) puts("Zle.");
    return 0;
}
```

```
D:\>Create2_Prog1Win xyz 345 987
D:\>Linia polecenia: 345 987
-----Program Prog1Win-----
argc= 3
xyz
345
987
sqrt(345) = 18.5742
sqrt(987) = 31.4166
Koniec Prog1Win
```

NOTATKI