

## 6. SYNCHRONIZACJA

Praca wątków winna być synchronizowana.

**1-szy wątek** modernizuje strukturę danych, z której korzysta **2-gi** wątek.

SO o sposobie pracy **1-go** i **2-go** wątku oraz zależnościach między nimi **nie ma** pojęcia.

Przydziela jednemu z nich określony **czas** CPU, a gdy dobiegnie **on** końca przerywa pracę wątku.

Gdy **2-gi** wątek odwoła się do struktury danych, której aktualizacja nie została zakończona, może to doprowadzić do nieprzewidzianych wyników.

**Błędna praca wątków może być niewidoczna, gdyż ich przerwanie musi wystąpić w określonej chwili, aby błąd w kodzie ujawnił się.**

Obiekt synchronizacji może znajdować się w stanie: -sygnalizowanym (*signaled state*)  
-niesygnalizowanym (*unsigaled state*).

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

**Sygnalizowany:** wątek/procesu zakończy pracę.

### 6.1. Funkcje Wait...

Funkcje **Wait...** powodują zawieszenie wykonania wątku do momentu zakończenia potomka.

#### 6.1.1. Funkcja WaitForSingleObject sygnalizowanie pojedynczego obiektu

```
int main()
{
    .....
    kod_1 (tworzy i aktywuje Wątek1, Wątek2, ...)
    .....
    flaga = WaitForSingleObject(Par1, Par2_Time);
    .....
    kod_2 // kod zawieszony przez Par1
    .....
}
```

**WaitForSingleObject** **zawiesza działanie Procesu/Wątku wywołującego** do momentu zakończenia wątku potomnego, wskazanego przez **Par1**.

Kod wyjścia wątku potomnego zwraca *GetExitCodeProcess*, po powrocie z **Wait...**

DWORD **WaitForSingleObject**(HANDLE **hObject**, DWORD **dwTimeOut**);

**hObject** -wskaźnik Obiektu, na którego zasygnalizowanie proces czeka.

**dwTimeOut** -czas oczekiwania [ms].

SO jest informowany, aby **nie wykonywał** wątku wywołującego funkcję **Wait...** aż do momentu upłynięcia czasu **dwTimeOut** lub zakończenia wskazanego procesu (w zależności od tego, co nastąpi pierwsze).

Wartość **INFINITE** (0xFFFFFFFF) wymusza oczekiwanie na zasygnalizowanie Obiektu.

Dla **dwTimeOut = 0**, funkcja sprawdzi stan Obiektu i natychmiast zwróci sterowanie.

Funkcja **Wait...** czeka na **zasygnalizowanie** Obiektu, wskazanego **1-szym** parametrem, gdy to nastąpi przekazuje sterowanie do wątku, który ją wywołał.

Obiekt jest **sygnalizowany** w momencie zakończenia jego procesów.

→ Jeżeli **sprawdzany** Obiekt znajduje się w stanie **niesygnalizowanym** system przełącza sterowanie do innego wątku.

→ **Wątek**, który **wywołał** funkcję **WaitForSingleObject** przechodzi w stan **Oczekiwania**, czeka aż wątek wskazany parametrem **hObject** zakończy działanie

→ Funkcja **Wait...** zwraca wartość wskazującą na przyczynę ponownego uruchomienia **wywołującego** funkcję wątku.

➤ Zwraca **WAIT\_TIMEOUT** (0x0102) jeżeli Obiekt nie zostanie zasygnalizowany zanim upłynie **dwTimeOut** (zwraca sterowanie, Obiekt pozostaje w stanie niesygnalizowanym).

➤ Zwraca **WAIT\_OBJECT\_0** (0x0000), gdy Obiekt, na który czekał wątek, został zasygnalizowany.

➤ Zwraca **WAIT\_FAILED** (0xFFFFFFFF), jeżeli wywołanie funkcji nie powiedzie się (np. niepoprawny uchwyt).

Więcej informacji o błędzie udostępnia wywołanie funkcji **GetLastError**.

Oczekując na **mutex**, funkcja może zwrócić wartość **WAIT\_ABANDONED** (0x0080) co oznacza, że: -nie został on zwolniony przez zakończony wątek, -system zwolnił **mutex** i pozwolił przejść go innemu wątkowi.

**Uwaga:** wartość **dwTimeOut = INFINITE** jest niebezpieczna.

Jeśli wskazany Obiekt nigdy nie zostanie zasygnalizowany, **wywołujący** funkcję wątek nie obudzi - pozostanie zablokowany, lecz nie będzie marnował czasu CPU.

Fragment kodu poniżej informuje system, aby nie wykonywał wątku wywołującego funkcję **Wait...** aż do zakończenia wskazanego procesu lub do upłynięcia 2000 ms.

```
DWORD idn = WaitForSingleObject(hProcess, 2000);
switch (idn) {
    case WAIT_OBJECT_0: // proces zakończył się
                        break;
    case WAIT_TIMEOUT: // proces nie zakończył się w ciągu 2 s.
                        break;
    case WAIT_FAILED: // błąd wywołania funkcji
                        break;
}
```

The **WaitForSingleObject** function can wait for the following objects:

- process,
- thread,
- event,
- mutex,
- timer,
- semaphore,
- console input .

Program **Watek6a** tworzy dwa wątki potomne, wypisujące odpowiednio 30 liter **A** oraz **\***.

Główny wątek (**main**) czeka - aż wątek **1** zakończy pracę - aby wypisać 30 liter **G**.

```
#include <iostream>
#include <windows.h>
#include <process.h>
using namespace std;
UINT WINAPI ZNAK(LPVOID);
void WatekGlowny(int, char);

int main() // Watek6a Oczekiwanie na zakończenie pracy wątku
{
    HANDLE hWatek1 = NULL, hWatek2 = NULL;
    UINT ID1 = 0, ID2 = 0;
    DWORD wynik1;
    char zn1 = 'A', zn2 = '*';

    // utworzenie i uruchomienie wątków potomnych i wiązanie ich uchwytów z nazwami hWatek1/2
    hWatek1 = (HANDLE)_beginthreadex(NULL, 0, ZNAK, &zn1, 0, &ID1);
    hWatek2 = (HANDLE)_beginthreadex(NULL, 0, ZNAK, &zn2, 0, &ID2);

    // WatekGlowny(300, '#');

    // czekanie aż watek1 zakończy działanie, czyli zostanie zasygnalizowany
    wynik1 = WaitForSingleObject(hWatek1, INFINITE);

    for(int i = 0; i < 30; i++) { // wypisuje 30 liter A co 80 ms
        cout << "G ";
        cout.flush();
        Sleep(80);
    }

    CloseHandle(hWatek1); CloseHandle(hWatek2);
    cout << "\nKoniec MAIN";
    // cin.get();
    return 0;
}

UINT WINAPI ZNAK(LPVOID znak)
{
    for(int i = 0; i < 30; i++) { // wypisuje 30 razy znak z parametru co 80 ms
        cout << *((char*)znak) << ' ';
        cout.flush();
        Sleep(80);
    }
    cout << endl;
}
```

A*	*A	A*	*A	A*	*A	A*	*A	A*	*A	A*	*A	A*	*A	A*	*A	A*	*A	A*	*A	A*	*A	A*	*A	A*	*A	A*	*A	A*	*A
A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*	A*
G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G

### Zadanie 6.1.

- 1 Zmodyfikować program **Watek6a** tak, aby znaki **zn1** i **zn2** mogły być wyświetlane w różnych ilościach.

- 2 Uaktywnić funkcję **WatekGlowny(300, '#');**

➤ Program czasami może działać niepoprawnie – wprowadzić stosowną modyfikację

### 6.1.2. Sygnalizowanie wielu obiektów

**Wątek Czeki aż zakończy pracę kilka jego wątków potomnych.**

DWORD **WaitForMultipleObjects**( DWORD **nCount**, CONST HANDLE \* **phObjects**,  
BOOL **bWaitAll**, DWORD **dwTimeOut** );

Funkcja wstrzymuje pracę **wywołującego** ją **wątku** (przenosi wątek w stan Oczekiwania), aż:

- **wszystkie** Obiekty jednocześnie przejdą w stan sygnalizowania,
- tylko **jeden** z nich osiągnie stan zasygnalizowania.

**nCount** - liczba obiektów, na których sygnalizowanie oczekujemy; wartość z przedziału od 1 do MAXIMUM\_WAIT\_OBJECTS (64).

**phObjects** -wskaźnik do Tablicy, zawierającej uchwytów obiektów; mogą być one różnego typu, ich liczba musi zgadzać się z podaną w pierwszym parametrze.

**bWaitAll** -**TRUE** funkcja zwróci sterowanie (do wątku wywołującego) wówczas, gdy **wszystkie** obiekty ulegną zasygnalizowaniu;  
-**FALSE** zwróci sterowanie po zasygnalizowaniu **jednego** z obserwowanych obiektów.

**dwTimeOut** -jak w przypadku funkcji *WaitForSingleObject* lecz dotyczy wielu obiektów.

Funkcja **WaitForMultipleObjects** *zwraca wartość*, określającą przyczynę wznowienia **wątku**, wywołującego tę funkcję.

- Funkcja zwraca wartość **WAIT\_FAILED** jeżeli jej wywołanie nie powiodło się.
- Zwraca **WAIT\_OBJECT\_0** dla **bWaitAll = TRUE** jeżeli doszło do zasygnalizowania wszystkich obiektów.
- Jeżeli **bWaitAll = FALSE** następuje powrót z funkcji po zasygnalizowaniu dowolnego obiektu.

Zwraca wartość należy do przedziału:

**WAIT\_OBJECT\_0 ÷ (WAIT\_OBJECT\_0 + nCount - 1)**

i jest indeksem uchwytu zasygnalizowanego obiektu z tablicy **phObjects**.

Jeżeli wartością zwracaną nie jest WAIT\_TIMEOUT ani WAIT\_FAILED, należy odjść od wartości zwracanej WAIT\_OBJECT\_0.

- Zwraca **WAIT\_TIMEOUT** dla **bWaitAll = FALSE** jeżeli po upływie czasu **dwTimeOut** wszystkie obiekty znajdują się w stanie niesygnalizowany.
- Zwraca **WAIT\_TIMEOUT** dla **bWaitAll = TRUE** jeżeli po upływie określonego czasu nie wszystkie obiekty weszły w stan sygnalizowania.

Gdy funkcja zwróci wartość z przedziału:

**WAIT\_ABANDONED\_0 ÷ (WAIT\_ABANDONED\_0 + nCount - 1)**

a parametr **bWaitAll** ustawiono na **TRUE**

to stan wszystkich obiektów jest zasygnalizowany i przynajmniej jeden z nich był **mutex'em**.

Gdy **bWaitAll** jest **FALSE** to zwrócona wartość pomniejszona o **WAIT\_ABANDONED\_0** (0x00000080) określa indeks pierwszego uchwytu, który dotyczy zasygnalizowanego **mutex'a**.

```
HANDLE H[3] = { Watek1, Watek2, Watek3 };
DWORD idn = WaitForMultipleObjects(3, H, FALSE, 2000);
switch (idn) {
    case WAIT_FAILED:           // błąd wywołania funkcji
                                break;
    case WAIT_TIMEOUT:          // żaden obiekt nie został zasygnalizowany w ciągu 2 s.
                                break;
    case WAIT_OBJECT_0 + 0:      // zakończył się wątek wskazywany przez H[0]
                                break;
    case WAIT_OBJECT_0 + 1:      // zakończył się wątek wskazywany przez H[1]
                                break;
    case WAIT_OBJECT_0 + 2:      // zakończył się wątek wskazywany przez H[2]
                                break;
}
```

**Watek6b** tworzy 3 wątki potomne (❶, ❷, ❸) o różnych czasach działania, wypisując litery: A, B, S.

Każdy wątek potomny generuje **swój własny wątek potomny** ④, wykonujący funkcję wątkową **ZNAK1()** .

GŁÓWNY wątek programu czeka aż praca dwóch wątków ❶, ❷ zostanie zakończona, po czym uaktywnia wątek ❸ oraz wypisuje w petli litery G.

Dwa wątki potomne (1,2) pracują równocześnie,  
zaś GŁÓWNY będzie kontynuował pracę dopiero po zakończeniu ich działania.

[illegible]

### Zadanie 6.2.

Wykonać modyfikację programu **Watek6b** , poprzez kombinacje czynności:

- zablokować wątek **hWatek1**
- zablokować wątek **hWatekA**
- zablokować wątek **hWatekB**
- zablokować funkcję **Wait..** w Głównym
- zablokować funkcję **Wait..** w Wątku

- Program czasami może działać niepoprawnie – wprowadzić stosowną modyfikację

### Zadanie 6.3.

Wykorzystując funkcję `Wait`, zmodyfikować program **Watek3a** aby działał poprawnie i **wydajnie**.

- Uwzględnić dwie wersje:
- tablice **A** i **B** zawierają po 30000 elementów
  - tablica **A** zawiera 20000 el. zaś tablica **B** zawiera 40000 el.

```

UINT WINAPI ZNAK(LPVOID);
UINT WINAPI ZNAK1(LPVOID);

    struct PARM {
        int n;
        char zn;
    };

int main()                                // Watek6b Oczekiwanie na zakończenie pracy 2-ch wątków
{
    HANDLE hWatekA = NULL, hWatekB = NULL, hWatekS = NULL;
    UINT ID_A = 0, ID_B = 0, ID_S = 0;
    PARM par1 = {20, 'A'}, par2 = {100, 'B'}, par3 = {30, 'S'};
        // uruchomienie 2-ch watkow potomnych
    hWatekA = (HANDLE)_beginthreadex(NULL, 0, ZNAK, &par1, 0, &ID_A);    // ❶
    hWatekB = (HANDLE)_beginthreadex(NULL, 0, ZNAK, &par2, 0, &ID_B);    // ❷
        HANDLE pWatek[2] = { hWatekA, hWatekB };                        // tablica wskaźników wątków

    // czekanie az jeden z watkow zawartych w tablicy pWatek zakonczy dzialanie
    DWORD ind = WaitForMultipleObjects(2, pWatek, FALSE, INFINITE);
        cout << "\n zakonczył watek idn=" << ind << endl;

    hWatekS = (HANDLE)_beginthreadex(NULL, 0, ZNAK, &par3, 0, &ID_S);    // ❸
        for(int i = 0; i < 30; i++) {                                    // wątek główny wypisuje 30 razy znak G
            cout << "G ";
            cout.flush(); Sleep(50);
        }

    CloseHandle(hWatekA); CloseHandle(hWatekB); CloseHandle(hWatekS);
    cout << "\nKoniec MAIN";
    // cin.get();
    return 0;
}

UINT WINAPI ZNAK(LPVOID par)
{
    PARM b*parm = (PARM*)par;
    intb M = parm->n;
    charb zn = parm->zn, zn1 = '1';

    HANDLE hWatek1;
    UINT ID1 = 0;

    hWatek1 = (HANDLE)_beginthreadex(NULL, 0, ZNAK1, &zn1, 0, &ID1);    // ❹
    WaitForSingleObject(hWatek1, INFINITE);

    for(int i = 0; i < 30; i++) {                                        // wypisuje 30 razy znak co M ms
        cout << zn << ' '; cout.flush();
        Sleep(M);
    } cout << endl;
}

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

```

## 6.2. Zdarzenia (Event)

synchronizacja w trybie **jądra**

**Zdarzenie** pozwala aby dany wątek poinformował inny o swej **gotowości** lub **wykonaniu** jakiegoś zadania.

Zdarzenie to **ObiektJądra** przyjmujący stan: sygnalizowany lub **niesygnalizowany**.

W stan sygnalizowany/niesygnalizowany wprowadzają specjalne funkcje.

Zadanie składa się z 2-ch części: 1-szą wykonuje **WątekA**, 2-gą **WątekB**.  
**Zdarzenia** używa w sytuacji, gdy **WątekA** ma wykonać 1-sze zadanie i chce zasygnalizować **WątkowiB**, aby wykonał 2-gie zadanie.

Na początku ObiektZdarzenie jest w stanie **niesygnalizowanym**.

Przechodzi w stan **sygnalizowany** po wykonaniu 1-go zadania przez **WątekA**.

Jest to sygnał dla **WątkaB**, aby wznowił działanie i wykonał 2-gą część zadania.

Istnieją dwa typy Zdarzeń:

**1. resetowane ręczne** : z chwilą zasygnalizowania zdarzenia tego typu, **wszystkie** wątki czekające na to zdarzenie zostają zaszeregowane do wykonania.

Ustawiany jest w stan **niesygnalizowany ręcznie**, za pomocą funkcji **ResetEvent** po jego sprawdzeniu.

**2. resetowane automatycznie**:

gdy zostaje zasygnalizowane ten typ zdarzenia, to **tylko jeden** z czekających na nie wątków zostaje zaszeregowany do wykonania.

Ustawiany jest w stan **niesygnalizowany** zaraz po ich **sprawdzeniu** przez funkcję *WaitForSingleObject* lub *WaitForMultipleObject*.

```
HANDLE CreateEvent(           // tworzy ObiektZdarzenie
LPSECURITY_ATTRIBUTES lpSa, // adres struktury z atrybutami zabezpieczeń ( NULL)
BOOL bManualReset,          // TRUE ręcznie sygnalizowane, FALSE automatycznie
BOOL bInitialState,         // stan początkowy: TRUE sygnalizowany, FALSE niesygnalizowany
LPCTSTR lpName              // nazwa ObiektuZdarzenie lub NULL
)
```

Funkcja zwraca względny uchwyt do utworzonego Obiektu (właściwy dla macierzystego procesu), lub **NULL** gdy utworzenie Obiektu nie powiedzie się.

Próba utworzenia Zdarzenia o **nazwie** już istniejącej nie zostanie zrealizowana, zaś funkcja zwróci uchwyt istniejącego;

parametry *bManualReset* i *bInitialState* nie są brane wówczas pod uwagę.

**GetLastError** zwróci ERROR\_ALREADY\_EXISTS (183).

Jeżeli podano nazwę istniejącego **semafora** lub **muteksu** wywołanie funkcji **CreateEvent** nie powiedzie się i zostanie zwrócona wartość **NULL**.

Kiedy ObiektZdarzenie nie będzie już potrzebny, należy wywołać funkcję *CloseHandle*.

Gdy utworzono wiele uchwytów, Obiekt zostanie zniszczony gdy wszystkie jego uchwytów zostaną zamknięte.

Fragment kodu tworzący automatycznie zasygnalizowane zdarzenie o nazwie "PAC" :

```
HANDLE hZ = NULL;
hZ = CreateEvent(NULL, FALSE, TRUE, "PAC");
```

Zmiana stanu zdarzenia na sygnalizowane:

BOOL **SetEvent**(HANDLE hEvent)

Funkcja zwraca wartość **nieszerowa**, gdy powiedzie się, w przeciwnym razie **zero**.

Zmiana stanu zdarzenia na **niesygnalizowany**:

BOOL **ResetEvent**(HANDLE hEvent)

Funkcja zwraca wartość **nieszerowa** gdy powiedzie się, w przeciwnym razie **zero**.

► Funkcja sygnalizuje zdarzenie i **natychmiast** ustawia je w stan **niesygnalizowany**:

BOOL **PulseEvent**(HANDLE hEvent)

Sygnalizuje ona tak długo wskazane zdarzenia aż wszystkie wątki, które na nie oczekują zostaną odblokowane.

Gdy to nastąpi zdarzenie jest ustawiane w stan **niesygnalizowany**.

Gdy żaden z wątków nie oczekuje na wskazane zdarzenie, funkcja zwraca sterowanie natychmiast.

Wywołanie funkcji dla zdarzenia resetowanego **ręcznie** powoduje wznowienie wszystkich wątków czekających na to zdarzenie.

Wywołanie dla zdarzenia resetowanego **automatycznie** sprawia, że tylko jeden z czekających wątków zostanie wznowiony (nie wiadomo, który).

Jeśli żaden wątek **nie czeka** na zasygnalizowanie zdarzenia, wywołanie nie daje żadnego efektu.

HANDLE **OpenEvent**(

DWORD **dwAccess**, // EVENT\_ALL\_ACCESS, EVENT\_MODIFY\_STATE

BOOL **bInherit**, // TRUE to proces utworzony przez CreateProcess dziedziczy uchwyt

LPCTSTR **lpName** // nazwa identyczna jak w **CreateEvent**

);

Zwraca uchwyt lub NULL w razie niepowodzenia

Efekt uboczny pomyślnego czekania na zdarzenie **resetowane automatycznie**:

Gdy wątek pomyślnie doczeka się takiego obiektu, zostanie on automatycznie ustawiony w stanie niesygnalizowanym.

Ponieważ system automatycznie cofa sygnalizację takiego ObiektuZdarzenia, z reguły nie wymaga on wywoływania funkcji **ResetEvent**.

Dla **resetowanego ręcznie**, nie zdefiniowano efektów ubocznych czekania.

Programy **Mapp1Parent** i **Mapp1Child** realizują przesyłanie komunikatów między Procesami poprzez plik mapowany w PAO.

Programy **Mapp1Parent** tworzy **Zdarzenie** niesygnalizowane nadając mu nazwę, zatem funkcja **Wait...** czeka na jego zasygnalizowanie (program zostaje wstrzymany).

Programy **Mapp1Child** otwiera **Zdarzenie**, następnie nadaje komunikat i ustawia **Zdarzenie** na sygnalizowane co wznowia działanie programu **Mapp1Parent**.

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

int main() // Mapp1Parent ← odbiera komunikaty od innego Procesu
{
HANDLE hMojeEvent = NULL; // uchwyt do obiektu Zdarzenie
HANDLE hMap = NULL; // uchwyt do obiektu reprezentującego plik zmapowany
char *pMap; // wskaźnik na obszar zmapowanego pliku
char Komunikat[55];
int sizeF = sizeof Komunikat;

//--- zdarzenia ręcznie sygnalizowanego, ze stanem początkowym niesygnalizowane
hMojeEvent = CreateEvent(NULL, TRUE, FALSE, "MojeEvent");
if(hMojeEvent==NULL) { printf("CreateEvent error:"); getchar(); return 1;}

HANDLE hFile =(HANDLE)0xffffffff; // = -1, mapowanie fragmentu pliku wymiany SO

//----- utworzenie Obiektu reprezentującego fragment zmapowany plik wymiany
hMap = CreateFileMapping(hFile, NULL,PAGE_READWRITE, 0, 1024, "MojMapp");
if(hMap==NULL) { printf("CreateFileMapping error:"); getchar(); return 2;}

//----- przydzielenie PAO na odwzorowanie pliku
pMap = (char *)MapViewOfFile(hMap, FILE_MAP_WRITE, 0, 0, 0);

//-----
do { WaitForSingleObject(hMojeEvent, INFINITE); //czeka na zasygnalizowanie Zdarzenia
    memcpy(Komunikat, pMap, sizeF); // odczyt ze zmapowanego pliku
    printf("\nOdebrano Komunikat : %s", Komunikat);
    if(!strcmp(Komunikat, "koniec")) break;
}
while(1);
UnmapViewOfFile(pMap);
CloseHandle (hMap);
return 0;
}
```

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

int main() // Mapp1Child --> nadaje komunikaty do innego Procesu
{
HANDLE hMojeEvent = NULL, hMapFile = NULL;
char *pMapFile, Komunikat[55];
int sizeF = sizeof Komunikat;

hMojeEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, "MojeEvent");
if(hMojeEvent==NULL) { printf("OpenEvent error:"); getchar(); return 1; }

if(!(hMapFile = OpenFileMapping(FILE_MAP_WRITE, FALSE, "MojMapp")))
{ printf("OpenFileMapping error: %d\n", GetLastError()); getchar(); return 2; }

pMapFile = (char *)MapViewOfFile(hMapFile, FILE_MAP_WRITE, 0, 0, 0);

//-----
do { printf("\nKomunikat dla obcego procesu : ");
    gets(Komunikat);
    memcpy(pMapFile, Komunikat, sizeF); // zapis do zmapowanego pliku
    PulseEvent(hMojeEvent); // sygnalizuje nadanie komunikatu (uaktywnia czekające wątki)
}
while(strcmp(Komunikat, "koniec"));

UnmapViewOfFile (pMapFile);
CloseHandle (hMapFile);

return 0;
}
```



Program **Zdarzenie1**

-**Wypełnia** tablicę znakami ASCII

-**Wyświetla** jej zawartość.

Działa w pętli. Naciśnięcie **Q** -KOŃCZY.

Program główny **Wyświetla**.

Wątek pochodny **Wypełniaj** wypełnia tablicę znakami.

Główny wątek CZEKA, aż **pochodny** wypełni tablicę, dopiero wtedy wyświetla jej zawartość.

Gdy główny wątek **Wyświetla**, wątek **pochodny** CZEKA na zakończenie jego pracy.

Programem sterują dwa zdarzenia **automatyczne**:

-zasygnalizowane **hZMain**, które zezwala na **Wypełnianie** tablicy znakami,

-niezasygnalizowane **hZWatek**, które zezwala na **Wyświetlanie** zawartości tablicy.

→ Wątek przed każdą modyfikacją tablicy **Oczekuje** na sygnalizację zdarzenia **hZMain**.

← Gdy wątek **Wypełni** tablicę znakami, zasygnalizowane jest zdarzenie **hZWatek**.

← Przed **Wyświetleniem** tablicy, Główny **Oczekuje** na sygnalizację zdarzenia **hZWatek**.

Gdy Główny Wyświetli tablicę, sygnalizuje zdarzenie **hZMain**, na które oczekuje wątek **Wypełniaj**.

Utworzenie Zdarzenia **hZMain** jako zasygnalizowane determinowane jest **bezpieczeństwem** rozruchu programu.

Bez tego główny wątek i pochodny uległyby zakleszczeniu, gdyż oczekiwałyby na siebie nawzajem. Każdy z nich sygnalizuje swoje ZDARZENIE dopiero wówczas, gdy drugi wątek dokona tego w **pierwszej** kolejności.

```
/* dwa ObiektyZdarzenia synchronizują dostęp w wspólnych Danych
   hZMain      -zezwala na Wypełnianie tablicy znakami
   hZWatek     -zezwala na Wyświetlanie zawartości tablicy */

#include <iostream>
#include <windows.h>
#include <process.h>
#include <conio.h>
using namespace std;
UINT WINAPI Wypełniaj(LPVOID);
const int MAX = 33;
HANDLE hZMain = NULL, hZWatek = NULL;
char tabC[MAX]; // tablica globalna

int main() // Zdarzenie1
{
    UINT IDW = 0;
    hZMain = CreateEvent(NULL, FALSE, TRUE, NULL); // sygnalizowane
    hZWatek = CreateEvent(NULL, FALSE, FALSE, NULL); // niesygnalizowane

    HANDLE hW = (HANDLE)_beginthreadex(NULL, 0, Wypełniaj, NULL, 0, &IDW);

    for(;;) {
        //czeka na zasygnalizowanie Zdarzenia hZWatek, tzn. az watek pochodny wypelni tabC
        WaitForSingleObject(hZWatek, INFINITE);
        for (int i = 0; i < MAX; i++) cout << tabC[i]; cout << endl; // Wyświetla
        if (kbhit() != 0) if (getch() == 'Q') break;
        Sleep(700); // przerwa między wypisywaniem wierszy
        SetEvent(hZMain); // sygnalizacja zdarzenia hZMain, na ktore oczekuje watek pochodny
    }
    CloseHandle(hW);
    CloseHandle(hZMain);
    return 0;
}

UINT WINAPI Wypełniaj(LPVOID parametr) // modyfikacja tablicy
{
    static char zn = 'A';
    while (1) {
        //-----oczekiwanie na zasygnalizowanie hZMain tzn. az watek glowny wypisze tablice
        WaitForSingleObject(hZMain, INFINITE);
        for(int i = 0; i < MAX; i++) tabC[i] = zn; // Wypełnia tablice
        SetEvent(hZWatek); // sygnalizowanie zdarzenia hZWatek, na ktore oczekuje GŁÓWNY
        if (zn < 'Z') zn++; else zn = 'A'; // zmienia znak wypełniający tablicę
    }
    _endthreadex(0);
}
```

## ❖ Przykład zastosowania Obiektu Zdarzenie ❖

Szkic programu **Zdarzenie** pokazuje metodykę synchronizowania 3-ch wątków w stosunku do czynności procesu macierzystego.

➔ Wątki **procesów**, które chcą uzyskać dostęp do danego obiektu, mogą:

- wywołać funkcję **CreateEvent** z tą samą wartością parametru **lpName**,
- wywołać funkcję **OpenEvent**.

❶ Na początku proces tworzy Zdarzenie resetowane **ręcznie** i **niesygnalizowane** z globalnym uchwytym.

Następnie uruchamiane są 3 wątki, każdy sięga do Danych:

**Zlicza:** zlicza liczby parzyste

**Suma:** oblicza sumę elementów

**Rozpietosc:** wyznacza rozpiętość zbioru

```
HANDLE hEvent;           // zmienna globalna typu UCHWYT
int main(void)           // Zdarzenie
{
    hEvent = CreateEvent(NULL, TRUE, FALSE, NULL); // resetowane ręcznie i niesygnalizowane
    HANDLE hTh[3];
    UINT ID1, ID2, ID3;
    hTh[0] = (HANDLE)_beginthreadex(NULL, 0, Zlicza, NULL, 0, &ID1);
    hTh[1] = (HANDLE)_beginthreadex(NULL, 0, Suma, NULL, 0, &ID2);
    hTh[2] = (HANDLE)_beginthreadex(NULL, 0, Rozpietosc, NULL, 0, &ID2);

    Operacje_na_pliku(); // np.: wczytanie zawartości pliku dyskowego do PAO
    SetEvent(hEvent);    // stan zaszygnalizowany umożliwia dostęp do PAO 3 wątkom
    return (0);
}

UINT WINAPI Zlicza(LPVOID param)
{
    WaitForSingleObject(hEvent, INFINITE); // czeka na zaszygnalizowanie Zdarzenia, co pozwoli
    { Działanie_na_bloku_PAO; }           // działać na pliku w PAO
    return (0);
}

UINT WINAPI Suma(LPVOID param)
{
    WaitForSingleObject(hEvent, INFINITE);
    { Działanie_na_bloku_PAO; }
    return (0);
}

UINT WINAPI Rozpietosc(LPVOID param)
{
    WaitForSingleObject(hEvent, INFINITE);
    { Działanie_na_bloku_PAO; }
    return (0);
}
```

**Wątek** potomny wywołuje funkcję *WaitForSingleObject*, która zawiesza go na czas działania funkcji wątku głównego.

Po odczytaniu danych główny wątek wywołuje funkcję **SetEvent**, która ustawia Zdarzenie w stan **sygnalizowany**;

w tym momencie **"każdy wątek"** dostaje CPU i dostęp do pamięci (wątki korzystają z pamięci w trybie ODCZYTU).

❷ Użycie Zdarzenia resetowanego **automatycznie** (**False**) zmieni zachowanie aplikacji.

Po wywołaniu **SetEvent** w głównym wątku, system wznowi wykonanie tylko **jednego** wątku pochodnego; nie wiadomo, którego, zaś pozostałe **dwa** muszą nadal **czekać**.

```
UINT WINAPI Zlicza(LPVOID param)
{
    WaitForSingleObject(hEvent, INFINITE);
    { Działanie_na_bloku_PAO; }
    SetEvent(hEvent);

    return(0);
}

UINT WINAPI Suma(LPVOID param)
{
    WaitForSingleObject(hEvent, INFINITE);
    { Działanie_na_bloku_PAO; }
    SetEvent(hEvent);

    return(0);
}

UINT WINAPI Rozpietosc(LPVOID param)
{
    WaitForSingleObject(hEvent, INFINITE);
    { Działanie_na_bloku_PAO; }
    SetEvent(hEvent);

    return (0);
}
```

► Zmodyfikowano funkcje wątkowe dodając przed powrotem **SetEvent()**.

Po zakończeniu analizy danych wątek wywołuje funkcję **SetEvent()**, co pozwala systemowi wznowić jeden z dwóch pozostałych wątków.

➔ Wątek, który zostanie zaszerogowany do wykonania, ma **wyłączny** dostęp do pamięci.

Nie wiadomo, który z nich zostanie wybrany, ale wiadomo, że kolejny wątek także otrzyma **wyłączny dostęp** do bloku pamięci.

Dopiero po jego zakończeniu można wznowić trzeci z wątków i zakończyć całą operację.

W przypadku **Zdarzenia** resetowanego **automatycznie** jest nieistotne, aby każdy z wątków pochodnych miał udostępniony blok pamięci tylko do Odczytu

**Zadanie.**

Uzupełnić szkic programu **Zdarzenie** do wersji wykonywalnej dla przypadku ❶ oraz ❷

Rozważyć modyfikacje treści programu wersji ❷ w stosunku do treści wersji ❶

### 6.3. Sekcja krytyczna (Critical Section) synchronizacja w trybie użytkownika

Sekcja krytyczna: **fragment kodu**, który musi uzyskać wyłączny dostęp do dzielonego zasobu.

➔ System może wywłaszczyć wątek działający w swojej sekcji krytycznej i aktywować inny.

**Nie aktywuje** wątku, który **chce** uzyskać dostęp do swojego zasobu dzielonego.

#### Tok postępowania:

Utworzyć strukturę typu:

**CRITICAL\_SECTION** csZasób;

Zainicjalizować składowe struktury:

**InitializeCriticalSection(&csZasób);**

Wszystkie odwołania do dzielonych zasobów umieścić między wywołaniami funkcji:

**EnterCriticalSection(& csZasób);**

**{ fragment kodu działający na zmiennych dzielonych }**

**LeaveCriticalSection(& csZasób);**

Obiekty CRITICAL\_SECTION można alokować jako **zmienne**:  
-globalne,  
-lokalne,  
-dynamiczne.

➔ Wątki korzystające z zasobu muszą znać adres chroniącej go zmiennej strukturalnej.

**Sekcje krytyczne nie nadają się do synchronizowania Działań w różnych Procesach.**

► Funkcja inicjalizująca składowe zmiennej strukturalnej:

VOID **InitializeCriticalSection**(LPCRITICAL\_SECTION **pcs**);

Funkcja tylko ustawia wartości składowych.

Funkcję trzeba wywołać, zanim którykolwiek z wątków wywoła **EnterCriticalSection**.

► **Należy** usunąć zmienną strukturalną, gdy wątki nie będą już korzystać z dzielonego zasobu, wywołując poniższą funkcję:

VOID **DeleteCriticalSection**(LPCRITICAL\_SECTION **pcs**);

W WINDOWS wątki czekające na sekcję krytyczną nigdy nie zostają zablokowane. Wywołanie **EnterCriticalSection** ulega przeterminowaniu, co powoduje zgłoszenie wyjątku. Czas [s], który musi upłynąć określa wartość **CriticalSectionTimeout** w podkluczu rejestru:

**HKEY\_LOCAL\_MACHINE\System\CurrentControlSet\Control\Session Manager**

Standardowo wynosi 2 592 000, czyli około 30 dni.

Nie należy ustawiać poniżej 3 sekund.

### Opis działania sekcji krytycznej

► Przed **kodem** korzystającym z dzielonego zasobu należy wywołać funkcję

VOID **EnterCriticalSection**(LPCRITICAL\_SECTION **pcs**);

Funkcja sprawdza czy aktualnie jakiś wątek korzysta z podanej sekcji krytycznej:

1. Jeśli żaden wątek nie korzysta z sekcji, funkcja zaznacza, żewołający wątek uzyskał dostęp do sekcji krytycznej i natychmiast wraca, umożliwiając wątkowi kontynuowanie działania.
2. Jeśli składowe struktury wskazują, żewołający wątek uzyskał dostęp do sekcji krytycznej, funkcja aktualizuje wskaźnik liczby dostępówwołającego wątku do sekcji i natychmiast wraca, umożliwiając wątkowi kontynuowanie działania.

Sytuacja ma miejsce tylko wtedy, gdy wątek wywoła **EnterCriticalSection** dwukrotnie z rzędu bez wywołania **LeaveCriticalSection**.

3. Jeśli struktura wskazuje, że **inny wątek** niż **wołający** ma **już** dostęp do sekcji krytycznej, funkcja **EnterCriticalSection** wprowadza **wołający** wątek w stan **Oczekiwania**.

➔ Wątek ten nie będzie marnował czasu CPU.

System **pamięta** o czekającym wątku i gdy aktualny proces używający sekcji krytycznej wywoła funkcję **LeaveCriticalSection**, automatycznie zaktualizuje składowe struktury i wznowi działanie zawieszono wątku.

Funkcja **EnterCriticalSection** działa prawidłowo, gdy **dwa** wątki wywołają ją jednocześnie w maszynie **jednoprocesorowej**:

- jeden z wątków uzyska dostęp,
- drugi przejdzie w stan **Oczekiwania**.

► Na końcu kodu korzystającego z zasobu dzielonego należy wywołać funkcję

VOID **LeaveCriticalSection**(PCRITICAL\_SECTION **pcs**);

Funkcja sprawdza wskazaną zmienną strukturalną **pcs** i zmniejsza o 1 wartość składowej zawierającej liczbę udostępnień sekcji wątkowi, który wywołał tę funkcję.

Jeśli wartość licznika jest **większa** od **0**, funkcja **nic nie robi** i wraca w miejsce wywołania.

Jeśli po odjęciu 1 wartość licznika jest równa 0, funkcja sprawdza, czy na sekcję nie czeka inny wątek (wywołanie **EnterCriticalSection**).

Jeśli istnieje taki wątek, funkcja aktualizuje składowe struktury CRITICAL\_SECTION i wznowia czekający wątek.

Jeśli żaden wątek **nie** czeka na sekcję, funkcja ustawia składowe struktury, aby wskazywały dostępność zasobu.



W programie **Critical1** : wątekA generuje liczby losowe  $\in (-9, -1)$ , zapisuje do **A** i wyświetla  
wątekB generuje liczby losowe  $\in (11.1, 99.9)$ , zapisuje do **A** i wyświetla  
Sekcja krytyczna chroni tablicę **A** przed nadpisywaniem

```
UINT WINAPI GEN(LPVOID);
void Disp(int, int, double *, char *);
void ObLiCzeNia(char, int);
double Generuj(float a, float b) {
    double w = (a + (b - a)*(double)rand()/RAND_MAX);
    for (long k=0; k < 1200; k++) log(pow((pow(sin(k)+1.1, 3.3)), 2.2)); // spowalnicz_1
    return floor(w * 100 + 0.5)/100; }

typedef struct PARM { float aa, bb; char zn; } *pPARM;

const int MAXDATA = 2000;
double B[4000], A[MAXDATA]; // zmienna dzielona to tablica A

CRITICAL_SECTION cs;

int main() // Critical1
{
    PARM parA = { -9.0, -1, 'A' }, parB = { 11.1, 99.9, 'B' };
    UINT ID1=0, ID2=0;
    InitializeCriticalSection(&cs);

    HANDLE hWA = (HANDLE)_beginthreadex(NULL, 0, GEN, &parA, 0, &ID1);
    HANDLE hWB = (HANDLE)_beginthreadex(NULL, 0, GEN, &parB, 0, &ID2);

    ObLiCzeNia('*', 200);

    cout << "\nKoniec programu Critical1"
    return 0;
}

UINT WINAPI GEN(LPVOID parametr)
{
    pPARM st = (pPARM)parametr;
    float oda = st->aa, dob = st->bb; char zn = st->zn;

    ObLiCzeNia(zn, 80);

    EnterCriticalSection( &cs );

    cout << "\n W sekcji krytycznej_" << zn << endl;
    for (int i = 0; i < MAXDATA; i++) {
        A[i] = Generuj(oda, dob);
    }
    Disp(0, 44, A, " ");
    LeaveCriticalSection( &cs );

    int k;
    double w = 0;
    ZeroMemory(B, 4000*sizeof(double));
    for (k=0; k < 4000 ; k++) B[k] = log(pow((pow(sin(k)+1.1,3.3)),2.2));
    for (k=0; k < 4000 ; k++) w += B[k];
    cout << "\nKoniec Watka." << zn << " w=" << w << endl;
}
```

```
void ObLiCzeNia(char zn, int n)
{
    double w;
    for (int k1=0; k1 < n; k1++) {
        for (long k2=0; k2<80000; k2++) w = cos(sin(pow(log(k2+1),2.2)));
        cout << zn;
    }
}
```

```
B*A**A*BA*AB*AB*BABBA*AB*A*B*ABBA*BB*A*AB*BA*BB*A*BA*BA**ABB*BB*BA*B*A*B*A*B*A*B
*A*BA*BAA*BABAAB*A*B*A*B*A*B*A*B*AB**AB*BAA*BB*AB**BA*A*B*A*B*AA*BABA*AA*BA**BA*
BABA*BA*BA*B**AB*BA*AB*BBAA*B*A*B*A*B*A*B*A*B*A*BA*BA*ABAAB*BAB*BAB*ABA*BAA*BAB**BA

W sekcji krytycznej_B
*****
11.22    61.15    28.27    82.92    63.05    53.72
42.21    90.66    84.17    77.4    26.57    87.38
74.2     56.71    38.1    12.44    19.23    43.47
24.19    25.84    98.88    50.68    21.68    11.52
11.9     44.66    58.32    61.83    64.54    * 65.02
25.87    69.98    51.14    42.37    16.17    65.07
80.66    82.37    57.27    37.92    88.89    75.63
95.98    93.3

W sekcji krytycznej_
Koniec WatkaAB
. W=-7228.17*****
-8.99    -4.55    -7.47    -2.61    -4.38    -5.21
-6.23    -1.92    -2.5    -3.1    -7.62    -2.21
-3.39    -4.94    -6.6    -8.88    * -8.28    -6.12
-7.84    -7.69    -1.19    -5.48    -8.06    -8.96
-8.93    -6.01    -4.8    -4.49    -4.25    -4.2
-7.69    -3.76    -5.44    -6.22    -8.55    -4.2
-2.81    -2.66    -4.89    -6.61    -2.08    -3.26
-1.45    -1.69

Koniec WatkaA. w=-7228.17*****
Koniec programu Critical1
```

### Zadanie 6.3.

Zmodyfikować program **Critical1** , poprzez różne kombinacje następujących czynności:

- zmieniać parametr **spowalnicz\_1**,
- zablokować sekcję krytyczną,
- zmieniać 2-gi parametr funkcji **ObLiCzeNia( )**;
- zmienić wartość **MAXDATA = 2000** na **MAXDATA = 40000**
- wstawić **spowalnicz\_1** do funkcji **Disp** z **k < 20000**

➡ **Program czasami może działać niepoprawnie – wprowadzić stosowną modyfikację**

### Zadanie 6.4.

Zmodyfikować program **Watek3a**, wykorzystując technikę **sekcji krytycznej**.

Celem modyfikacji jest optymalizacja działania programu, połączona z zagwarantowaniem pierwszeństwa generowania danych nad ich sortowaniem.

### 6.3.1. Uwagi

❑ Zamiast *EnterCriticalSection* można użyć:

```
BOOL TryEnterCriticalSection(PCRITICAL_SECTION pcs);
```

Funkcja **nie** przenosi wywołującego ją wątku w stan Oczekiwania.

➔ Jeśli sekcja jest akurat zajęta przez inny wątek, funkcja zwraca **FALSE**.

W każdym innym przypadku zwraca wartość TRUE.

Wątek może szybko sprawdzić, swoje szanse na dostęp do zasobu dzielonego i jeśli niema, może kontynuować działanie, zamiast przechodzić w stan Oczekiwania.

Gdy funkcja zwraca **TRUE**, zaznacza w składowych struktury CRITICAL\_SECTION udostępnienie zasobu nowemu wątkowi.

Każde wywołanie tej funkcji, które zwraca **TRUE**, musi być uzupełnione wywołaniem funkcji *LeaveCriticalSection*.

Struktura CRITICAL\_SECTION zdefiniowana jest w pliku WinNT.h jako RTL\_CRITICAL\_SECTION.

Struktura RTL\_CRITICAL\_SECTION jest typem zdefiniowanym w WinBase.h.

#### ❑ Blokada wirowa

➤ Aktywna tylko dla maszyn wieloprocesorowych.

Wątek próbujący wejść do sekcji krytycznej zajętej przez inny wątek, przechodzi natychmiast w stan **Oczekiwania**; zmienia swój tryb wykonania z użytkownika na tryb jądra (około 1000 cykli CPU).

Windows umożliwia włączenie **blokad wirowej** do obsługi sekcji krytycznych.

Z chwilą wywołania *EnterCriticalSection* funkcja uruchamia pętlę, w której próbuje **określoną liczbę** razy uzyskać dostęp do zasobu.

Gdy próby zakończą się niepowodzeniem, następuje przejście na tryb jądra i wejście w stan Oczekiwania.

Blokada wirowa w sekcji krytycznej, wymaga zainicjalizowania sekcji funkcją:

```
BOOL InitializeCriticalSectionAndSpinCount(PCRITICAL_SECTION pcs, DWORD dwSpinCount);
```

**dwSpinCount** -liczba prób dostępu do zasobu, które należy wykonać przed przejściem wątku w stan oczekiwania.

Dowolna liczba z przedziału (0, 0x00FFFFFF).

W maszynie jednoprosesorowej **dwSpinCount = 0**, gdyż wielokrotne próby nie mają sensu: wątek kontrolujący zasób nie zwolni go, jeśli w tym czasie inny wątek będzie, co chwila sprawdzał stan tego zasobu.

```
DWORD SetCriticalSectionSpinCount(PCRITICAL_SECTION pcs, DWORD dwSpinCount);
```

**dwSpinCount** jest ignorowany, jeśli aplikacja wykonuje się w komputerze jednoprosesorowym.

### ❑ Wiele sekcji krytycznych

Jeśli aplikacja zawiera **kilka** niepowiązanych ze sobą dzielonych zasobów, efektywniej jest utworzyć dla każdego z nich **oddzielną** zmienną typu CRITICAL\_SECTION.

```
CRITICAL_SECTION cs_A;  
CRITICAL_SECTION cs_C;  
UINT WINAPI FuncThread(LPVOID par)  
{  
    int i;  
    EnterCriticalSection(&cs_A);  
    for (i = 0; i < MAX; i++) tabA[i] = Generuj(a, b);  
    LeaveCriticalSection(&cs_A);  
  
    // Ciąg DALSZY KODU  
    EnterCriticalSection(&cs_C);  
    for (i = 0; i < MAX; i++) tabC[i] = pow(i, 2.2);  
    LeaveCriticalSection(&cs_C);  
}
```

#### ➤ Zakleszczenie

Gdy trzeba korzystać z dwóch zasobów jednocześnie, implementacja może wyglądać jak przedstawiono obok.

W wątkach **FuncThread1** i **FuncThread2** może dojść do zakleszczenia funkcji *EnterCriticalSection* i *LeaveCriticalSection*.

```
CRITICAL_SECTION cs_A;  
CRITICAL_SECTION cs_C;  
UINT WINAPI FuncThread1(LPVOID par)  
{  
    EnterCriticalSection(&cs_A);  
    EnterCriticalSection(&cs_C);  
    for (int i = 0; i < MAX; i++) tabA[i] = pow(tabC[i], a);  
    LeaveCriticalSection(&cs_A);  
    LeaveCriticalSection(&cs_C);  
}
```

Zaczyna wykonywać się funkcja **FuncThread1** i wchodzi w posiadanie sekcji krytycznej **cs\_A**. Następnie funkcja **FuncThread2** dostaje CPU i wątek wchodzi w posiadanie sekcji krytycznej **cs\_C**. Żadna z funkcji **FuncThread1** i **FuncThread2** nie może wejść w posiadanie potrzebnego jej zasobu dzielonego.

```
CRITICAL_SECTION cs_A;  
CRITICAL_SECTION cs_C;  
UINT WINAPI FuncThread2(LPVOID par)  
{  
    EnterCriticalSection(&cs_C);  
    EnterCriticalSection(&cs_A);  
    for (int i = 0; i < MAX; i++) tabA[i] = log10(tabC[i]);  
    LeaveCriticalSection(&cs_A);  
    LeaveCriticalSection(&cs_C);  
}
```

➔ Należy zawsze prosić o dostęp do zasobów dokładnie w takiej samej kolejności.

➔ Podczas wywołania funkcji *LeaveCriticalSection* kolejność nie ma znaczenia, gdyż funkcja ta nie przenosi wątku w stan Oczekiwania.

## 6.4. Mutex synchronizacja w trybie jądra

**Mutex** (*mutual exclusion* -wzajemne wykluczanie) to **Obiekt Jądra**, gwarantujący **wątkowi** wyłączny dostęp do pojedynczego zasobu.

Odpowiada pojęciu semafor binarny.

Służą do ochrony bloków pamięci dostępnych dla wielu wątków (lub procesów).

**Mutex'y gwarantują, że wątek, który uzyskał dostęp do bloku pamięci, ma go wyłącznie na swój użytek.**

Z jednego **mutexu** mogą korzystać wątki **różnych procesów** (poprzez nazwę mutexa).

Wątek może podać limit czasu czekania na uzyskanie dostępu do zasobu (**Wait...**(mutex, TimeOut)).

**W danej chwili tylko jeden wątek może przejąć mutex, inny może zrobić to wówczas, gdy mutex zostanie zwolniony.**

**Mutex** pozwala zapobiegać wielokrotnemu uruchomieniu aplikacji przez użytkownika.

**Mutex** ulega **zasygnalizowaniu** gdy **nie** jest przejęty przez żaden wątek.

**Mutex** zawiera: -tradycyjny licznik użyć,  
-**ID** wątku,  
-licznik rekursji.

**ID** wątku -wskazuje, który wątek jest aktualnie **właścicielem mutexu**,

jeśli **ID == 0**, to **mutex nie należy** do żadnego wątku i jest w stanie **sygnalizowanym**.

jeśli **ID ≠ 0** to **mutex** jest własnością wątku i jest **niesygnalizowany**.

**licznik rekursji** -zlicza **rekurencyjne** wejścia wątku w posiadanie określonego **mutex'a**.

Funkcja tworząca **mutex**:

HANDLE **CreateMutex**(

LPSECURITY\_ATTRIBUTES **lpSa**, // adres SECURITY\_ATTRIBUTES (NULL)

BOOL **InitialOwner**, // stan początkowy

LPCTSTR **lpName** // nazwa mutex'u lub NULL

);

Funkcja zwraca uchwyt utworzonego **mutexu**, lub NULL w przypadku nie powodzenia

**InitialOwner** -wartość **FALSE** zeruje **ID** wątku i licznik rekursji;

**mutex nie należy** do żadnego wątku i jest **sygnalizowany**.

-dla wartości **TRUE**, **ID** wątku przyjmie wartość identyfikatora wątku wołającego funkcję **CreateMutex**, a licznik rekursji wartość **1**;

• ponieważ **ID ≠ 0** **mutex** będzie na początku **niesygnalizowany**

• Wątek tworzący **mutex** staje się od razu jego właścicielem.

Jeżeli tworzony **mutex** już istnieje, (jest zarejestrowany w systemie o takiej samej nazwie), funkcja nie tworzy nowego mutexu, zwraca uchwyt istniejącego,

funkcja **GetLastError** zwróci wartość **ERROR\_ALREADY\_EXISTS(183)**.

Proces może uzyskać uchwyt **istniejącego** już **mutexu** poprzez funkcję:

```
HANDLE OpenMutex(DWORD dwAccess, // access flag: MUTEX_ALL_ACCESS
                  BOOL blnInheritHandle, // inherit flag
                  LPCTSTR lpName // nazwa mutex'u
                  );
```

➔ Po zakończeniu korzystania z zasobu wątek powinien zwolnić **mutex**, funkcją **ReleaseMutex**.

➔ Po zwolnieniu przechodzi w stan sygnalizowania.

```
BOOL ReleaseMutex( HANDLE hMutex );
hMutex -uchwyt zwalnianego mutexu.
ReleaseMutex jest odpowiednikiem Sygnalizuj(mutex)
```

Funkcja zwraca wartość różną od zera w przypadku powodzenia, w innym razie zero.

Zwraca zero gdy wątek wywołujący **ReleaseMutex** nie posiada wskazanego **mutexu**.

Funkcja zmniejsza licznik rekursji mutexu o 1.

Jeśli wątek uzyskał **mutex** kilka razy z rzędu, musi tyle samo razy wywołać **ReleaseMutex** (aby wyzerować licznik rekursji).

Gdy to nastąpi, **ID** wątku w **mutexie** przyjmie wartość 0 i **mutex** zmieni stan na sygnalizowany.

Wątek uzyskuje dostęp do dzielonego zasobu za pomocą wywołania funkcji **Czekaj()** i przekazania do niej uchwytu **mutex'u** strzegącego zasobu.

Funkcja **Czekaj()** sprawdza w obiekcie jądra, czy **ID** wątku ma wartość 0 (czy wątek sygnalizowany) i jeśli tak, ustawia pole **ID** na identyfikator wołającego wątku, licznik rekursji na **1** i pozostawia wątek wołający w stanie **Wykonywania**.

Jeśli funkcja **Czeka()** stwierdzi, że **ID** wątku w obiekcie jądra ma wartość różną od zera (**mutex** jest niesygnalizowany), wołający wątek przechodzi w stan **Oczekiwania**.

Gdy **ID** wątku w **mutexu** wróci do 0, ustawi **ID** na identyfikator czekającego wątku, licznik rekursji na **1** i zezwoli **Oczekującemu** wątkowi wznowić działania.

➔ Wątek przejmuje **wolny mutex** za pomocą funkcji:

```
WaitFor...(hmutex, TimeOut)
```

zaś przejęty **mutex** przechodzi w stan **niesygnalizowany**.

Inne wątki oczekujące na ten sam **mutex** są Blokowane

(nie wychodzą z **WaitFor...(hmutex, ...)**).

➔ Wątek posiadający **mutex** może przejąć go wielokrotnie, musi później wielokrotnie zwolnić.

Gdy **mutex** nie jest potrzebny, aplikacja powinna go zniszczyć funkcją **CloseHandle**.

Gdy **WĄTEK** otrzyma **mutex**, ma pewność, że uzyskał wyłączny dostęp do chronionego zasobu.

Inne wątki, próbujące sięgnąć do tego zasobu (czekające na ten sam **mutex**), przejdą w stan **Oczekiwania**

**Mutex powinien synchronizować tylko fragmenty kodu wątków / procesów**

Działanie programu sygnalizują markery:

<b>'1', 'a'</b>	(wątekA);
<b>'2', 'b'</b>	(wątekB);
<b>'*'</b>	(wątek główny).

```
UINT WINAPI DispB(LPVOID tekst)
{
    LPSTR txt = (LPSTR)tekst;
    for(int i=0; i < MAX; i++) {
        ObLiCzeNia('2', 40);
        WaitForSingleObject(hMutex, INFINITE);
        for (int k=0; (t = *(txt + k)); k++) {
            for (long k2=0; k2<90000; k2++) cos(sin(pow(log(k2+1),2.2)));
            cout << t;
        }
        ReleaseMutex(hMutex);
        ObLiCzeNia('b', 40);
    }
    cout << "\n koniec2 \n";
}
```

- Program czasami może działać niepoprawnie – wprowadzić stosowną modyfikację

Zmodyfikować program **Mutex1**, aby działał identycznie z jedną funkcją wątkową **Disp()**.



W programie **Mutex1a** wątek główny i 2 wątki potomne (utworzone na bazie jednej funkcji), wyświetlają odpowiednio kolejne znaki tekstu:

**główny**, **AAAAA**, **BBBBBBBBB**.

**Mutex** pilnuje aby znaki tekstów nie przeplatały się.

➔ Wyświetlać znaki może tylko ten wątek, który przejmie **mutex**.

Funkcja **WaitForSingleObject** czeka 5000 ms na sygnalizację **mutexu**, po czym zwraca wartość.

Gdy zwróci **WAIT\_TIMEOUT** oznacza, że upłynął czas oczekiwania, w którym **mutex** nie został zasygnalizowany, co generuje odpowiedni komunikat i kończy awaryjnie program.

Naciśnięcie klawisza **Q** przerywa pracę głównej pętli **while(1)** w funkcji **main**, co powoduje zniszczenie obiektu **mutex** i zakończenie pracy głównego wątku (zakończenie wątków potomnych).

Pierwsza wyświetlana jest tablica **txtG** (należy do **main**).

Utworzony sygnalizowany **mutex** wchodzi do **Wait...** programu głównego.

- natychmiast opuszcza **Wait...** ze statusem **niesygnalizowany**, działanie wątków zostaje przez **mutex** zablokowane
- program główny realizuje cykl **Wyświetlania**
- po zakończeniu Wyświetlania **mutex** zostaje zwolniony; któryś z wątków może rozpocząć Wyświetlanie swoich treści.

```
głównyAAA*AAA*AA*BBB*BBB*BBB*BBB*AAA*AAA*AA*BBB*BBB*BBB*BBB*AA*A
AAA*AA*BBB*BBB*BBB*BBB*AA*AAA*AAA*BBB*BAAAA*AAA*AB*BBB*BBB*BB
B*BA*AAA*AAA*AB*BBB*BBB*BBB*BA*AAA*AAA*AB*BBB*BBB*BBB*BA*AAA
*AAA*AB*BBB*BBB*BA*AAA*AB*BBB*BBB*BBB*BA*AAA*AAA*AB*BBB*BBB*B
BB*BA*AAA*AAA*AB*BBB*BBB*BBB*BA*AAA*AAA*AB*BBB*BBB*BBB*B*AB*
BBB*BBB*BBB*BA*AAA*AAA*AB*BBB*BBB*BBB*BA*AAA*AAA*AB*BBB*BBB*B
BB*B A*AAA*AAA*A B*BBB*BBB*BBB*B A*AA
Koniec programu Mutex1a.
```

### Zadanie 6.6.

Zmodyfikować program **Mutex1a**, poprzez różne kombinacje następujących czynności:

- zablokować **spawalniacz\_1**,
- zablokować **spawalniacz\_2**,
- zmienić parametry **spawalniaczy**,
- zablokować semafor,
- zmienić 2-gi parametr funkcji **ObLiCzeNia('\*', 44)**;

➔ Program czasami może działać niepoprawnie – wprowadzić stosowną modyfikację

### Zadanie 6.7.

Zmodyfikować program **Watek3a**, wykorzystując możliwości **mutex'a**.

Celem modyfikacji jest optymalizacja działania programu, połączona z zagwarantowaniem pierwszeństwa generowania danych nad ich sortowaniem.

```
#include <iostream>
#include <windows.h>
#include <process.h>
#include <conio.h>
#include <cmath>
using namespace std;
UINT WINAPI Disp(LPVOID);
char txtG[] = "głównyProgramWatek",
     txtA[] = "AAAAAAAAAAAA",
     txtB[] = "BBBBBBBBBBBBBBBB";
HANDLE hMutex = NULL;

int main(void) // Mutex1a
{
    UINT ID_A=0, ID_B=0;
    DWORD wynikWait = 0;

    hMutex = CreateMutex(NULL, FALSE, NULL); // sygnalizowany mutex, czyli wolny
    HANDLE hWatekA = (HANDLE)_beginthreadex(NULL, 0, Disp, txtA, 0, &ID_A);
    HANDLE hWatekB = (HANDLE)_beginthreadex(NULL, 0, Disp, txtB, 0, &ID_B);

    for (long k=0; k < 5000000; k++) (1.1/2.2); // spawalniacz_1

    while(1) {
        //czekanie 5s. na zasygnalizowanie mutexu, do momentu aż zostanie zwolniony przez wątek A lub B
        ❶ wynikWait = WaitForSingleObject(hMutex, 5000);
        if(wynikWait == WAIT_TIMEOUT) { cin.get(); cout << "Nie przechwycono mutexu \n"; return -1; }

        ❷ for (int i = 0; txtG[i] != '\0'; i++) cout << txtG[i] << flush;
        if (kbhit()!=0) if (getch() == 'Q' ) break;
        for (long k=0; k < 7000000; k++) pow(sin(1.1),log(3.3)); // spawalniacz_2

        ❸ ReleaseMutex(hMutex); // zwolnienie mutexu po wyświetleniu txtG

        ObLiCzeNia('*', 44);
    }
    CloseHandle(hWatekA);
    CloseHandle(hWatekB);
    CloseHandle(hMutex);
    cout << "\nKoniec programu Mutex1a.";
    // cin.get();
    return 0;
}

UINT WINAPI Disp(LPVOID tekst)
{
    LPSTR t, txt = (LPSTR)tekst;
    while(1) {
        WaitForSingleObject(hMutex, INFINITE);
        for (int i = 0; *(t = (txt + i)); i++) {
            cout << *t;
            Sleep(30); // co będzie po zablokowaniu
        }
        ReleaseMutex(hMutex);
    }
    return 0;
}
```



## ❑ Mutex – podsumowanie

Po zasygnalizowaniu **mutex'a** SO sprawdza, czy na niego nie czekają inne wątki.

**Jeśli tak**, wybiera jeden z czekających wątków i przydziela mu **mutex**; ustawia **ID** wątku w **mutex'ie** na identyfikator wybranego wątku i zwiększa licznik rekursji do wartości 1.

**Jeśli nie** ma wątku czekającego na zwolniony **mutex**, pozostaje on w stanie sygnalizowanym -gotowy do natychmiastowego przejścia przez **pierwszy** wątek, który zgłosi się.

### ► Niech wątek próbuje wejść w posiadanie **niesygnalizowanego mutexu**.

W takim przypadku wątek zazwyczaj przechodzi w stan **Oczekiwania**.

SO sprawdza, czy wątek próbujący przejść **mutex** nie ma **takiego samego** identyfikatora, jaki występuje w polu **ID mutexu**.

Jeśli tak, SO pozwala wątkowi dalej się wykonywać - mimo że **mutex** był **niesygnalizowany**.

Po pomyślnym otrzymaniu **mutexu** przez wątek, SO zwiększa licznik rekursji tego **mutexu** o 1.

Wątek czekający na ten sam **mutex** kilka razy z rzędu, doprowadzi do ustawienia licznika rekursji tego **mutexu** na wartość większą niż 1.

## ❑ Wątek porzucony

**Tylko mutexy przechowują informację, który wątek otrzymał go.**

Pozwala to wątkowi wejść w posiadanie **mutexu** wtedy, gdy **nie** jest on sygnalizowany.

Wyjątek ten odnosi się także do wątków, które usiłują **mutex** zwolnić.

Gdy wątek wywołuje funkcję **ReleaseMutex**, funkcja sprawdza, czy **ID** wołającego wątku pokrywa się z **ID** wątku w **mutexie**.

Jeśli tak, licznik rekursji jest zmniejszany.

Jeśli oba **ID** są różne, **ReleaseMutex** nic nie robi i zwraca do wątku FALSE (znak niepowodzenia).

Wywołanie w takiej sytuacji **GetLastError** daje błąd ERROR\_NOT\_OWNER.

### ► Wątek posiadający **mutex** zakończy działanie (*ExitThread*, *TerminateThread*, *ExitProcess* lub *TerminateProcess*) i nie zwolnił **mutexu**.

Czy **mutex** będzie osiągalny dla innych wątków ?

SO potraktuje taki **mutex** jako porzucony

- wątek, który wszedł w jego posiadanie, nie będzie mógł już nigdy go zwolnić, gdyż przestał istnieć.

SO śledzi wszystkie wątki i **mutexy**, wie kiedy dochodzi do porzucenia wątku.

Wtedy automatycznie zeruje **ID** wątku w **mutexie** oraz jego licznik rekursji, a następnie sprawdza, czy jakiś wątek nie czeka na ten **mutex**.

Jeśli czeka, wybierze jeden z czekających wątków, ustawi **ID** wątku w **mutexie** (na identyfikator wybranego wątku), licznik rekursji na 1 i ustawi do wykonania.

Efekt nie różni się od zwykłego zwolnienia **mutexu**, z tym że funkcja **Czekająca** nie zwraca do wątku wartości powrotnej **WAIT\_OBJECT\_0**, lecz specjalną wartość **WAIT\_ABANDONED**.

Wartość ta (stosowana tylko w przypadku mutexów) wskazuje, że **mutex**, na który czekał wątek, należał do innego wątku zakończonego przed zwolnieniem dzielonego zasobu.

## 6.5. Czasomierz (Timer)

### Obiekt Jądra

Obiekt samoczynnie sygnalizowany w określonym momencie lub w regularnych odstępach czasu.

Umożliwia podjęcie działań o ustalonej godzinie.

**Obiekty Czasomierze tworzone są zawsze w stanie niesygnalizowanym.**

Można czekać na zasygnalizowanie czasomierza, przekazując jego uchwyt do funkcji **WaitFor...**

Funkcja tworząca czasomierz:

HANDLE **CreateWaitableTimer**(

LPSECURITY\_ATTRIBUTES lpTimerAttribute

BOOL fManualReset, // TRUE sygnalizowane ręcznie  
// FALSE sygnalizowane automatycznie

LPCTSTR lpName

);

Zasygnalizowanie z **fManualReset** = TRUE wznawia wszystkie czekające na niego wątki.

Zasygnalizowanie z **fManualReset** = FALSE wznawia tylko jeden z czekających wątków.

Moment sygnalizowania czasomierza ustawia funkcja **SetWaitableTimer**.

BOOL **SetWaitableTimer**(

HANDLE hTimer, // uchwyt do ustawianego czasomierza,  
const LARGE\_INTEGER \*pDueTime, // ma się uaktywnić po raz pierwszy  
LONG lPeriod, // jak często ma powtarzać aktywację [ms]  
PTIMERAPCROUTINE pfnCompletionRoutine, // zazwyczaj NULL  
PVOID pvArgToCompletionRoutine, // zazwyczaj NULL  
BOOL fResume, // zazwyczaj FALSE  
);

**lPeriod** [ms] podaje jak często ma być sygnalizowany czasomierz po pierwszej aktywacji.

Interwał 2-ch godzin odpowiada liczbie: 2 godzin \* 60 minut \* 60 sekund \* 1000 ms.

Jeżeli **lPeriod** ma wartość 0 to czasomierz uaktywni się tylko raz.

**pfnCompletionRoutine**, **pvArgToCompletionRoutine**

wykorzystywane są przy organizacji asynchronicznych wywołań procedur (*Asynchronous Procedure Call* - **APC**)

W momencie uaktywnienia czasomierza wstawia on do kolejki wątek, który wywołał **SetWaitableTimer**.

Zazwyczaj ustawione są na **NULL**, co oznacza że z chwilą nadejścia wskazanego momentu stan czasomierza ma się zmienić na **sygnalizowany**.

**fResume** przydatny gdy komputery potrafią zawieszać i wznawiać swoje działanie.

Pisząc aplikację, w której chcesz przypominać o terminach spotkań za pomocą czasomierzy, należy ustawić parametr na TRUE.

Po uaktywnieniu czasomierza przerywa on stan zawieszenia komputera (jeśli był on zawieszony) i budzi wątki czekające na ten czasomierz.

Gdy parametr **fResume** jest ustawiony na **FALSE**, to po zasygnalizowaniu czasomierza żaden z czekających na niego wątków nie zostanie po obudzeniu przydzielony CPU, dopóki nie nastąpi wznowienie pracy komputera przez użytkownika.

➔ Można **nie** przekazywać do **SetWaitableTimer** **bezwzględnej** godziny pierwszej aktywacji.

Można użyć wartości względnej, przekazując przez parametr **pDueTime** liczbę **ujemną**, która zostanie potraktowana jako liczba **100-nanosekundowych** interwałów.

1 s = 1000 ms = 1 000 000 µs = 10 000 000 interwałów 100-ns.

```
// ustawienie czasomierza na pierwszą aktywację w 5 sekund po wywołaniu SetWaitableTimer
LARGE_INTEGER li;
HANDLE hTimer = CreateWaitableTimer(NULL, FALSE, NULL); // resetowany automatycznie
const int nano100 = 100000000; // jednostką czasomierza jest 100 ns.
li.QuadPart = -(5 * nano100); // ujemny czasu, gdyż chodzi o czas względny
SetWaitableTimer(hTimer, &li, 2 * 60 * 60 * 1000, NULL, NULL, FALSE);
```

Po skorzystaniu z czasomierza należy:

zamknąć go funkcji **CloseHandle**,

lub ponownie wywołać **SetWaitableTimer** z nowymi kryteriami aktywacji.

Względny uchwyt istniejącego czasomierza udostępnia funkcja **OpenWaitableTimer**.

HANDLE **OpenWaitableTimer**(

DWORD dwDesiredAccess,  
BOOL bInheritHandle,  
LPCTSTR lpName  
);

Funkcja wyłączająca aktywację czasomierza:

BOOL **CancelWaitableTimer**(HANDLE hTimer);

Aby zmienić same kryteria czasomierza, nie trzeba wywoływać funkcji **CancelWaitableTimer**, a dopiero potem **SetWaitableTimer**.

➔ Każde kolejne wywołanie **SetWaitableTimer** automatycznie unieważnia wcześniejszą specyfikację momentu sygnalizacji czasomierza.

W programie **Czasomierz1** wątki pochodne oparte na funkcji **Licz()**.

Wątek główny i pochodne wykonują obliczenia, sygnalizując je poprzez drukowanie: **^**, **\***, **A**, **B**.

Wątek **A** uruchamiany jest po **5** s.,

wątek **B** po **10** s.,

wątek **główny** po **15** s.

Czasomierz generuje czas względny, w stosunku do uruchomienia - ***SetWaitableTimer***

[illegible]

```

UINT WINAPI Licz(LPVOID parametr)
{
    pPARM st = (pPARM)parametr;
    char zn= st->zn;
    int n= st->nn;

    double w = 0;

    for (int k1=0; k1 < n; k1++) {
        for (long k2=0; k2 < 110000; k2++)  w += cos(sin(pow(log(k2+1),2.2)));
        cout << zn;
    }
}

```

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

#define Czas cout<<endl; GetTimeFormat(NULL,0,NULL,NULL,buf,255); cout<<buf<<endl;

UINT WINAPI Licz(LPVOID);

typedef struct PARM {
    char zn;
    int nn;
} *pPARM;

const int nano100 = 10000000;

int main() // Czasomierz1      Czasomierz względny
{
    PARM parmA = {'A', 50}, parmB = {'B', 50}, parmG = {'*', 50}, parm0 = {'^', 990};
    DWORD wynikWait = 0;
    UINT ID0=0, ID1=0, ID2=0;
    HANDLE hWA = NULL, hWB = NULL, hTimer;
    LARGE_INTEGER li;
    char buf[256];

    _beginthreadex(NULL, 0, Licz, &parm0, 0, &ID0);
    hTimer = CreateWaitableTimer(NULL, TRUE, NULL); // resetowanie ręczne

    li.QuadPart = -5 * nano100; // względny czas uruchomienia
    SetWaitableTimer(hTimer, &li, 0, NULL, NULL, FALSE);      Czas
    wynikWait = WaitForSingleObject(hTimer, INFINITE);
    if (wynikWait == WAIT_OBJECT_0)
        hWA = (HANDLE)_beginthreadex(NULL, 0, Licz, &parmA, 0, &ID1);

    li.QuadPart = -10 * nano100;
    SetWaitableTimer(hTimer, &li, 0, NULL, NULL, FALSE);      Czas
    wynikWait = WaitForSingleObject(hTimer, INFINITE);
    if (wynikWait == WAIT_OBJECT_0)
        hWB = (HANDLE)_beginthreadex(NULL, 0, Licz, &parmB, 0, &ID2);

    li.QuadPart = -20 * nano100;
    SetWaitableTimer(hTimer, &li, 0, NULL, NULL, FALSE);      Czas
    wynikWait = WaitForSingleObject(hTimer, INFINITE);
    if (wynikWait == WAIT_OBJECT_0) Licz(&parmG);      Czas

    CloseHandle(hWA);
    CloseHandle(hWB);
    CloseHandle(hTimer);
    cout << "KONIEC.";
    return (0);
}
```

### Czasomierz o bezwzględnym czasie aktywacji

Windows bases system time on Coordinated Universal time UTC.

UTC-based time is loosely defined as the current date and time of day in Greenwich, England

```

/* stawienie czasomierza na pierwszą aktywację w dniu 1 stycznia 2006 roku
o godzinie 12.00, a później równo co 2 godziny. */
HANDLE          hTimer;           // deklaracje zmiennych lokalnych.
SYSTEMTIME      st;
FILETIME        ftLocal, ftUTC;
LARGE_INTEGER liUTC;

hTimer = CreateWaitableTimer(NULL, FALSE, NULL); // resetowanie automatyczne
// parametry pierwszego zasygnalizowania
st.wYear        = 2015;           // rok
st.wMonth       = 7;              // Lipiec
st.wDayOfWeek   = 0;             // bez znaczenia
st.wDay         = 1;             // pierwszy dzień miesiąca
st.wHour        = 12;            // 12.00
st.wMinute      = 0;             // zerowa minuta godziny
st.wSecond      = 0;            // zerowa sekunda minuty
st.wMilliseconds = 0;           // zerowa milisekunda sekundy

SystemTimeToFileTime(&st, &ftLocal);

LocalFileTimeToFileTime(&ftLocal, &ftUTC); // konwersja czasu lokalnego na UTC.

liUTC.LowPart = ftUTC.dwLowDateTime; // konwersja FILETIME na LARGE_INTEGER
liUTC.HighPart = ftUTC.dwHighDateTime;

SetWaitableTimer(hTimer, &liUTC, 2 * 60 * 60 * 1000, NULL, NULL, FALSE);

```

Najpierw następuje inicjalizacja struktury SYSTEMTIME, wskazującej moment pierwszej aktywacji (zasygnalizowania) czasomierza.

Drugi parametr funkcji `SetWaitableTimer` ma w prototypie postać `const LARGE_INTEGER*` co uniemożliwia użycie bezpośrednio struktury `SYSTEMTIME`.

Struktury FILETIME i LARGEINTEGER mają identyczny format binarny, lecz różnią się sposobem wyrównania w pamięci. Adresy struktur FILETIME muszą zaczynać się od 32-bitowej granicy, natomiast adresy struktur LARGEINTEGER od 64-bitowej granicy.

Wywołanie *SetWaitableTimer* z użyciem struktury FILETIME zadziała prawidłowo tylko wtedy, gdy struktura ta zaczyna się od granicy 64-bitowej.

Kompilator gwarantuje wyrównanie początku struktury LARGEINTEGER do granicy 64-bitowej.

Bezpieczniej jest skopiować składowe FILETIME do składowych LARGEINTEGER i użyć tej ostatniej w wywołaniu *SetWaitableTimer*.

**UWAGA** Procesory **x86** same radzą z odwołaniami do niewyrównanych danych.

Przekazanie adresu FILETIME do *SetWaitableTimer* zadziała dla aplikacji uruchomionej na platformie x86.

Inne procesory zachowują się inaczej, zgłaszając wyjątek, który powoduje zakończenie procesu i wygenerowanie błędu: `EXCEPTION DATATYPE MISALIGNMENT`.

Funkcja *SetWaitableTimer* wymaga podawania czasu w formacie UTC (Coordinated Universal Time).

Konwersje realizuje funkcja *LocalFileTimeToFileTime*.

W programie **Czasomierz2** wątek **A** uruchamiany jest przez Czasomierz o **bezwzględnym** czasie aktywacji. Wątki wykonują obliczenia, sygnalizując poprzez drukowanie:  $\wedge * A, B$ .

```

UINT WINAPI Licz(LPVOID);

typedef struct PARM {
    char zn;
    int nn;
} *pPARM;

HANDLE Czasomierz(WORD Min, WORD Godz, WORD Dz=18)
{
    HANDLE          hTimer;
    SYSTEMTIME      st;
    FILETIME        ftLocal, ftUTC;

    hTimer = CreateWaitableTimer(NULL, FALSE, NULL);           // resetowanie automatyczne

    st.wYear       = 2015;                                     // parametry pierwszego zasygnalizowania
    st.wMonth       = 7;
    st.wDayOfWeek   = 0;
    st.wDay         = Dz;
    st.wHour        = Godz;
    st.wMinute      = Min;
    st.wSecond      = 0;
    st.wMilliseconds = 0;

    SystemTimeToFileTime(&st, &ftLocal);

    LocalFileTimeToFileTime(&ftLocal, &ftUTC);                //działa poprawnie tylko na płaszczyźnie x86

    SetWaitableTimer(hTimer, (LARGE_INTEGER*)&ftUTC, 0, NULL, NULL, FALSE);

    cout << "Czekam na "<< st.wHour<<":"<<st.wMinute << "..."<<endl;

    return (hTimer);
}

int main(void)                                // Czasomierz2 - o bezwzględnym czasie aktywacji
{
    HANDLE hTimerA;
    UINT ID0=0; ID1=0, ID2=0;
    HANDLE hWA = NULL, hWB = NULL;
    DWORD wynikWait = 0;
    PARM parmA = {'A', 50}, parmB = {'B', 50}, parmG = {'*', 70}, parmG = {'^', 70};

    _beginthreadex(NULL, 0, Licz, &parm0, 0, &ID0);

    hTimerA = Czasomierz(47, 22, 18);

    wynikWait = WaitForSingleObject(hTimerA, INFINITE);

    if (wynikWait == WAIT_OBJECT_0)
        hWA = (HANDLE) _beginthreadex(NULL, 0, Licz, &parmA, 0, &ID1);

    hWB = (HANDLE) _beginthreadex(NULL, 0, Licz, &parmB, 0, &ID2);

    Licz(&parmG);

    CloseHandle(hTimerA);
    CloseHandle(hTimerA);

    cout << "KONIEC.";
    //cin.get();
    return (0);
}

```

## ANEX 6.1 Semafor (Semaphore)

## synchronizacja w trybie jądra

Semafor to Obiekty Jądra przeznaczone do zliczania zasobów.

Zawierają:

- tradycyjny licznik użyć,
- maksymalny licznik zasobów**, maksymalna liczba zasobów kontrolowana przez semafor,
- aktualny licznik zasobów**, liczba aktualnie kontrolowanych zasobów.

Bufor serwera może pomieścić jednocześnie żądania 3-ch klientów.  
Inicjalizacja procesu serwerowego tworzy pulę 3-ch wątków, do obsługi żądań klientów.  
Na początku nie ma żądań, więc serwer **blokuje** wykonanie wszystkich wątków.  
Gdy nadejdą 2 żądania, serwer **wznawia** wykonanie 2 wątków do obsługi klientów.  
• Użycie semafora pozwala monitorować i wznawiać wątki z puli.  
**Maksymalny licznik** zasobów należy ustawić na 3.  
**Aktualny licznik** zasobów należy ustawić na 0.  
  
Pojawianie się kolejnego żądania zwiększy **aktualny licznik** o 1, a po uruchomieniu wątku obsługującego to żądanie - zmniejszany o 1.

Zasady pracy z semaforami:

1. Jeśli **aktualny licznik** zasobów ma wartość **większą** od 0, semafor jest **sygnalizowany**; po przejściu przez wątek zasobu wartości licznika zmniejszana jest o 1.
2. Jeśli **aktualny licznik** zasobów ma wartość równą 0, semafor jest **niesygnalizowany** i żaden inny wątek nie może go przejąć.
3. **Aktualny licznik** zasobów nigdy nie przyjmuje wartości ujemnej.
4. Wartość **aktualnego licznika** zasobów nie może być większa maksymalnej.
5. Wątek niepotrzebujący już semafora powinien go zwolnić (zwiększyć jego licznik).

Cecha semaforów:

operacja Sprawdzenia-Ustawienia realizowana jest atomowo, tzn. że operacja ta nie może zostać przerwana przez żaden inny wątek.

HANDLE **CreateSemaphore**(

LPSECURITY_ATTRIBUTE	lpSa,	
LONG	lInitialCount,	// ile zasobów używanych jest na wstępie
LONG	lMaximumCount,	// maksymalna liczba zasobów
LPCTSTR	lpName	// nazwa semafora lub NULL

);

Funkcja zwraca uchwyt do utworzonego semafora lub NULL gdy wywołanie nie powiedzie.

Funkcja nie wykona żadnej operacji (tylko zwróci uchwyt istniejącego semafora) przy próbie utworzenia semafora o takiej samej nazwie jak już istniejący w systemie.

Wywołanie funkcji GetLastError (zwraca ostatni błąd) zwróci wartość ERROR\_ALREADY\_EXISTS.

HANDLE hsemafor = **CreateSemaphore**(NULL, 0, 3, NULL);

Ponieważ **aktualny licznik** zasobów ma wartość 0, semafor jest niesygnalizowany.

W efekcie każdy wątek czekający na semafor pozostaje w stanie **Oczekiwania**.

Wątek uzyskuje dostęp do zasobu wywołując funkcję czekającą **Wait... (hsemafor)**.

Funkcja **Wait** sprawdza **aktualny licznik** zasobów.

Jeżeli jego wartość jest **większa** od 0 (semafor sygnalizowany), zmniejsza ją o 1, i wznawia wywołujący wątek.

Dopiero po zmniejszeniu **aktualnego licznika** zasobów system pozwala innemu wątkowi zażądać dostępu do zasobu.

Jeżeli jego wartość jest równa 0 (semafor niesygnalizowany), system umieści wywołujący wątek w stanie **Oczekiwania**.

Kiedy inny wątek zwiększy **aktualny licznik** zasobów semafora, system pozwoli czekającemu wątkowi wznowić działanie (zmniejszając jednocześnie bieżący licznik zasobów).

Proces może uzyskać własny, względny uchwyt istniejącego już semafora:

HANDLE **OpenSemaphore**(

DWORD	fdwAccess,	// flagi dostępu
BOOL	blnInheritHandle,	// gdy TRUE to zwrócony uchwyt będzie dziedziczony
LPCTSTR	lpName	

);

Niepotrzebny semafor zwalniamy wywołując funkcję:

BOOL **ReleaseSemaphore**( HANDLE hsem, LONG lReleaseCount, LPLONG plPreviousCount);

Funkcja zwraca wartość różną od zera w przypadku powodzenia, lub zero

**hsem** -uchwyt semafora, który zwalniamy.

**lReleaseCount** -wartość o jaką będzie zwiększony **aktualny licznik** zasobów semafora, najczęściej jest to liczba 1, ale nie jest to konieczne.

Funkcja nie wykona żadnego działania i zwróci FALSE, jeżeli dodanie wartości

**lReleaseCount** do aktualnego stanu licznika spowoduje przekroczenie zakresu,

**plPreviousCount** adres zmiennej w której zostanie zapisana poprzednia wartość aktualnego licznika semafora; można podać NULL.

Zwolnienie semafora nie powoduje jego zniszczenia, przełącza go jedynie w stan sygnalizowania.

Aby zniszczyć obiekt semafora trzeba zamknąć jego uchwyt funkcją **CloseHandle()**.



Program **Semafor1** tworzy semafor, ustalając maksymalną i początkową wartość licznika na 3.

Oznacza to, że zaraz po jego utworzeniu program może trzykrotnie przejąć semafor.

Pętla **while(1)** czeka na sygnalizację semafora,

gdy się doczeka wypisuje tekst "**odwołanie\_Nr**",  
gdzie **Nr** jest numerem sygnalizacji semafora.

Klawisze (1 ÷ 4) umożliwiają zwolnienie semafora przez zwiększenie jego licznika  
( klawisz 1 zwiększa o 1, klawisz 2 zwiększa o 2, itd.)

Po naciśnięciu **klawisza 4** program podejmuje próbę zwiększenia licznika semafora o 4,  
→ nie powiedzie się, gdyż licznik przekroczyłby maksymalną wartość **3**.

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

int main(void) // Semafor1
{
    char zn;
    HANDLE hSemafor = NULL;
    DWORD hWynik = 0;
    WORD LicznikOdwołan = 0;
    hSemafor = CreateSemaphore(NULL, 3, 3, NULL);
    while(1) { // wypisuje tekst gdy uda się przejąć semafor
        // oczekiwanie na sygnalizację semafora, nastąpi to gdy wartość jego licznika > 0
        hWynik = WaitForSingleObject(hSemafor, 50);
        // jeżeli oczekiwanie na semafor zakończyło się sukcesem funkcja zwraca WAIT_OBJECT_0,
        // jego licznik jest zmniejszany o jeden; wypisywany jest tekst określający numer przejęcia semafora
        if(hWynik == WAIT_OBJECT_0) printf("odwołanie%i ", LicznikOdwołan++);
        if(kbhit() != 0) {
            switch(zn = getch()) {
                case 'Q': goto KONIEC;
                // jeżeli naciśnięto 1 zwalniamy semafor, zwiększając jego licznik o 1
                case '1': ReleaseSemaphore(hSemafor, 1, NULL); break;
                // jeżeli naciśnięto 2 zwalniamy semafor zwiększając jego licznik o 2
                case '2': ReleaseSemaphore(hSemafor, 2, NULL); break;
                case '3': ReleaseSemaphore(hSemafor, 3, NULL); break;
                case '4': ReleaseSemaphore(hSemafor, 4, NULL); break;
            }
        }
    }
    KONIEC: CloseHandle(hSemafor);
    puts("KONIEC");
    return 0;
}
```

odwołanie0 odwołanie1 odwołanie2 odwołanie3  
odwołanie4 odwołanie5 odwołanie6 odwołanie7  
odwołanie8 odwołanie9 odwołanie10 KONIEC

Pytanie: 3 pierwsze teksty wypisane zostały bez  
wciskania klawisza → dlaczego ?