Programowanie współbieżne

Tworzenie i obsługa semaforów oraz wątków przy użyciu funkcji Windows API.

Cel zadania.

Celem zadania jest poznanie podstawowych funkcji Windows API umożliwiających tworzenie i obsługę obiektów semaforów oraz wątków w systemie Windows.

Treść zadania.

Należy zapoznać się z podstawowymi funkcjami umożliwiającymi, w przypadku semaforów:

- tworzenie oraz zamykanie obiektów semaforów,
- podnoszenie semaforów,
- wstrzymywanie procesów na semaforach.

w przypadku wątków natomiast:

- uruchamianie wątków,
- zawieszanie wątków,
- i zatrzymywanie wątków.

Pytania kontrolne.

1) Jaka jest różnica pomiędzy wątkiem, a procesem?
2) Omówić funkcje API operujące na semaforach.

Część teoretyczna.

I. Tworzenie i obsługa semaforów za pomocą funkcji Windows API.

1. Wstęp.

Semafory **S** są obiektami niskiego poziomu umożliwiającymi synchronizację współdziałających procesów, rozwiązanie problemu wzajemnego wykluczania oraz innych zagadnień programowania współbieżnego. Semafory zawierają zmienną całkowitą przyjmującą wartości od zera do pewnej określonej wartości maksymalnej. Licznik ten jest dekrementowany zawsze gdy proces oczekuje na semaforze, natomiast inkrementowany w momencie zasygnalizowania pewnego zdarzenia, tzw. zwolnienia semafora poprzez dowolny proces. Instrukcje Czekaj(**S**) i Sygnalizuj(**S**) są instrukcjami atomowymi, tzn. żadna instrukcja nie może być wpleciona między sprawdzanie stanu licznika, a jego dekrementację. W Windows API czekanie na semaforze umożliwia nam funkcja WaitForSingleObject(), natomiast zwolnienie semafora zapewnia ReleaseSemaphore().

Zanim jednak możemy używać obiektów semaforów, należy je utworzyć przy użyciu funkcji CreateSemaphore(), której parametrami są między innymi początkowa wartość licznika oraz wartość maksymalna którą licznik może osiągnąć. W momencie gdy obiekt semafor nie jest już dłużej potrzebny, należy go zamknąć przy użyciu funkcji CloseHandle().

2. Opis funkcji API.

CreateSemaphore

Funkcja CreateSemaphore tworzy semafor o określonej nazwie lub semafor bezimienny.

HANDLE **CreateSemaphore**(LPSECURITY_ATTRIBUTES **atrybuty**,
LONG **wartosc_poczatkowa**, LONG **wartosc_maksymalna**,
LPCSTR **nazwa**);

Parametry:

atrybuty – wskaźnik do struktury SECURITY_ATTRIBUTES określającej, czy zwracany uchwyt może być odziedziczony przez proces potomny. Może być NULL, wówczas uchwyt nie będzie odziedziczony.

wartosc_ poczatkowa – określa początkową wartość licznika semafora. Wartość ta musi być większa lub równa zero oraz mniejsza bądź równa wartości maksymalnej.
wartosc_maksymalna – określa maksymalną wartość licznika semafora. Musi być większe od zera.

nazwa – to wskaźnik do ciągu znaków zakończonych zerem określającego nazwę semafora. Długość tego ciągu jest ograniczona poprzez wartość MAX_PATH, a sam ciąg natomiast nie może zawierać znaków: backspace oraz '\'. Jeżeli to pole jest wskaźnikiem NULL, to utworzony zostaje semafor bez nazwy.

Wartość zwracana:

Jeżeli semafor zostanie utworzony, to zwracany jest uchwyt do tego semafora. W przeciwnym wypadku zwracana jest wartość NULL.

Przykład:

```
HANDLE semafor_binarny = CreateSemaphore(NULL, 0, 1, NULL);
```

Powyższa konstrukcja może posłużyć do utworzenia semafora binarnego.

ReleaseSemaphore

Funkcja ReleaseSemaphore zwiększa licznik semafora o określoną wartość. Jeżeli jakiś proces oczekuje na podniesienie semafora, to po wywołaniu tej funkcji, zostanie wznowiony.

```
BOOL ReleaseSemaphore(HANDLE semafor,
                      LONG wzrost,
                      LPLONG poprzednia_wartosc);
```

Parametry:

semafor – identyfikuje obiekt semafor. Jest to wartość zwrócona przez funkcję CreateSemaphore().

wzrost – określa wartość o jaką zwiększony zostanie jednorazowo licznik semafora. Wartość ta musi być większa od zera. Jeżeli zwiększenie licznika o podaną wartość spowodowałoby wykroczenie poza maksymalną wartość, to licznik nie zostanie zmieniony, a funkcja ReleaseSemaphore() zwróci wartość FALSE.

poprzednia_wartosc – to wskaźnik do zmiennej 32 bitowej, w której zapisana zostanie poprzednia wartość licznika (przed inkrementacją). Może być równe NULL.

Wartość zwracana:

Jeżeli wywołanie funkcji powiedzie się to TRUE. W przeciwnym wypadku FALSE.

Przykład:

```
ReleaseSemaphore(semafor_binarny, 1, NULL);
```

Wywołanie tej funkcji spowoduje zwiększenie licznika semafora utworzonego w poprzednim przykładzie o 1.

WaitForSingleObject

Wywołanie tej funkcji spowoduje zatrzymanie procesu, dopóki jeden z poniższych warunków nie zostanie spełniony:

- określony obiekt zostanie zasygnalizowany
- upłynie określony czas oczekiwania

DWORD **WaitForSingleObject**(HANDLE **obiekt**, DWORD **czas_oczekiwania**);

Parametry:

obiekt – charakteryzuje obiekt. Między innymi może być uchwytem do semafora utworzonego poprzez CreateSemaphore().

czas_oczekiwania – czas w milisekundach, po upływie którego działanie zatrzymanego procesu zostanie wznowione. Wartość INFINITE oznacza oczekiwanie do skutku.

Wartość zwracana:

Jeżeli wywołanie powiedzie się, to zwracana wartość określa zdarzenie, które spowodowało powrót z funkcji:
WAIT_OBJECT_0 – spełniony pierwszy warunek,
WAIT_TIMEOUT – spełniony warunek drugi jeśli czas_oczekiwania różny od INFINITE.
W przeciwnym wypadku zwracana jest wartość WAIT_FAILED.

Przykład:

```
WaitForSingleObject(semafor,INFINITE);
```

Powyższe wyrażenie powoduje wstrzymanie procesu na semaforze identyfikowanym poprzez uchwyt semafor. Powrót z funkcji WaitForSingleObject() nastąpi w momencie, gdy semafor zostanie podniesiony.

CloseHandle

Funkcja CloseHandle() zamyka obiekt identyfikowany poprzez uchwyt.

BOOL **CloseHandle**(HANDLE **obiekt**);

Parametry:

obiekt – identyfikator obiektu.

Wartość zwracana:

Jeżeli wywołanie funkcji powiedzie się to TRUE, w przeciwnym wypadku FALSE.

Przykład:

```
CloseHandle(semafor);
```

Powyższa funkcja zamyka obiekt semafor utworzony w pierwszym przykładzie.

3. Niezbędne pliki nagłówkowe oraz biblioteki.

CreateSemaphore	-	winbase.h	kernel32.lib
ReleaseSemaphore	-	winbase.h	kernel32.lib
WaitForSingleObject	-	winbase.h	kernel32.lib
CloseHandle	-	winbase.h	kernel32.lib

II. Tworzenie i obsługa wątków przy użyciu funkcji Windows API.

1. Wstęp.

Wątek jest podstawowym bytem, któremu przydzielana jest moc procesora. Wszystkie wątki uruchomione w obrębie pewnego procesu dzielą tę samą przestrzeń adresową, zmienne globalne oraz zasoby systemowe danego procesu.

2. Opis funkcji API.

CreateThread

Funkcja CreateThread() tworzy wątek w celu uruchomienia go w przestrzeniu adresowej procesu macierzystego.

```
HANDLE CreateThread(
            LPSECURITY_ATTRIBUTES atrybuty,
            DWORD rozmiar_stosu,
            LPTHREAD_START_ROUTINE funkcja,
            LPVOID argument,
            DWORD flaga,
            LPDWORD identyfikator
      );
```

Parametry:

atrybuty – wskaźnik do struktury SECURITY_ATTRIBUTES określającej, czy zwracany uchwyt może zostać odziedziczony przez proces potomny, czy też nie. Może przyjmować wartość NULL, wówczas uchwyt nie będzie odziedziczony.

rozmiar_stosu – określa rozmiar stosu w bajtach. Jeżeli podamy zero, to utworzony stos będzie miał ten sam rozmiar co stos procesu macierzystego.

funkcja – adres funkcji WINAPI, której prototyp wygląda w następujący sposób:

```
DWORD WINAPI Funkcja(LPVOID);
```

argument – określa 32 bitową wartość przekazywaną bezpośrednio nowemu wątkowi.

flaga – zawiera dodatkowe flagi kontrolujące proces tworzenia nowego wątku. Jeżeli użyjemy flagi CREATE_SUSPENDED, to wątek po utworzeniu zostanie zawieszony, aż do czasu wywołania funkcji ResumeThread(). Jeżeli flaga przyjmie wartość zero, to wątek zostanie uruchomiony natychmiast po utworzeniu.

identyfikator – wskaźnik do zmiennej 32 bitowej służącej do przechowania identyfikatora wątku.

Wartość zwracana:

Jeżeli wywołanie funkcji powiedzie się to zwrócona wartość będzie uchwytem do nowo utworzonego wątku, jeżeli nie to zwrócona zostanie wartość NULL.

ResumeThread

Funkcja ResumeThread() dekrementuje licznik zawieszeń wątku. Jeżeli licznik ten osiągnie wartość zero, to działanie wątku zostaje wznowione.

```
DWORD ResumeThread(
        HANDLE uchwyt
    };
```

Parametry:

uchwyt – zawiera uchwyt do wątku którego działanie ma zostać wznowione.

Wartość zwracana:

Jeżeli wywołanie funkcji powiedzie się, to zwracana jest ostatnia wartość licznika zawieszeń wątku. W przeciwnym wypadku zwracana jest wartość 0xFFFFFFFF.

SuspendThread

Funkcja SuspendThread() zawiesza wykonywanie wątku określonego poprzez parametr uchwyt. Ponadto inkrementowany jest licznik zawieszeń procesu.

```
DWORD SuspendThread(
        HANDLE uchwyt
    );
```

Parametry:

uchwyt – określa uchwyt wątku który ma zostać zawieszony.

Wartość zwracana:

Jak w przypadku funkcji ResumeThread().

SetThreadPriority

Funkcja SetThreadPriority() ustala priorytet określonego wątku. Wartość ta, włącznie z priorytetem klasy procesu, określa bazowy priorytet wątku.

```
BOOL SetThreadPriority(
    HANDLE  uchwyt,
    int priorytet
    );
```

Parametry:

uchwyt – identyfikuje wątek.

priorytet – określa priorytet wątku. Może przyjmować jedną z poniższych wartości:

THREAD_PRIORITY_ABOVE_NORMAL	wskazuje priorytet o 1 większy niż priorytet klasy.
THREAD_PRIORITY_BELOW_NORMAL	wskazuje priorytet o 1 niższy niż priorytet klasy.

THREAD_PRIORITY_HIGHEST	wskazuje priorytet o 2 większy niż priorytet klasy.
THREAD_PRIORITY_LOWEST	wskazuje priorytet o 2 niższy niż priorytet klasy.
THREAD_PRIORITY_NORMAL	wskazuje normalny priorytet, taki jak dla klasy procesu.

Wartość zwracana:

Jeżeli akcja się powiedzie to zwracana jest wartość większa od zera, w przeciwnym razie zero.

GetThreadPriority

Funkcja ta zwraca priorytet danego wątku.

int **GetThreadPriority**(
HANDLE **uchwyt**
);

Parametry:

uchwyt – identyfikuje wątek.

Wartość zwracana:

Jeżeli wywołanie funkcji powiedzie się to zwracana jest wartość priorytetu określonego wątku. W przeciwnym razie zwracana jest wartość THREAD_PRIORITY_ERROR_RETURN.

ExitThread

Funkcja ExitThread() zakańcza dany wątek.

VOID **ExitThread**(
DWORD **kod_wyjsciowy**
};

Parametry:

kod_wyjsciowy – określa kod wyjściowy. Wartość tego kodu dla zakończonego wątku można odczytać za pomocą funkcji GetExitCodeThread().

TerminateThread

Funkcja ta zatrzymuje wątek.

BOOL **TerminateThread**(
HANDLE **uchwyt**,
DWORD **kod_wyjsciowy**
);

Parametry:

uchwyt – identyfikuje wątek, który ma zostać zatrzymany.

kod_wyjsciowy – określa kod wyjściowy. Wartość tego kodu dla zakończonego wątku można odczytać za pomocą funkcji GetExitCodeThread().

Wartość zwracana:

W przypadku udanego zakończenia działania wątku zwracana jest wartość TRUE. W przeciwnym wypadku natomiast FALSE.

CloseHandle

Funkcja CloseHandle() zamyka obiekt identyfikowany poprzez uchwyt.

BOOL **CloseHandle**(HANDLE **obiekt**);

Parametry:

obiekt – identyfikator obiektu.

Wartość zwracana:

Jeżeli wywołanie funkcji powiedzie się to TRUE, w przeciwnym wypadku FALSE.

3. Niezbędne pliki nagłówkowe oraz biblioteki.

CreateThread	-	winbase.h	kernel32.lib
ResumeThread	-	winbase.h	kernel32.lib
SuspendThread	-	winbase.h	kernel32.lib
SetThreadPriority	-	winbase.h	kernel32.lib
GetThreadPriority	-	winbase.h	kernel32.lib
ExitThread	-	winbase.h	kernel32.lib
TerminateThread	-	winbase.h	kernel32.lib
CloseHandle	-	winbase.h	kernel32.lib

III. Przykładowa aplikacja.

Poniższy kod jest próbą rozwiązania problemu producenta – konsumenta przy użyciu wątków oraz semaforów zaimplementowanych za pomocą funkcji Windows API.

```
#include <windows.h>
#include <time.h>

static HINSTANCE        hInstApp;
static HWND             hwndApp;
char                   szAppName[]="Problem producenta - konsumenta.";

static HANDLE           theElementy;
static HANDLE           theMiejsca;
static HANDLE           theProducent;
static HANDLE           theKonsument;

static int              we_indeks       = 0;
static int              wy_indeks       = 0;
static int              N               = 20;
static char*            bufor           = NULL;

//---------------------------------------------------------------------

DWORD WINAPI Producent(LPVOID)
{
        while(1)
        {
                Sleep((rand()%10)*100 + 100);
```

```
                WaitForSingleObject(theMiejsca,INFINITE);
                bufor[we_indeks] = 1;
                we_indeks = (we_indeks + 1) % N;
                ReleaseSemaphore(theElementy,1,NULL);
                RedrawWindow(hwndApp,NULL,NULL,RDW_INTERNALPAINT);
        };
        return 0;
};

DWORD WINAPI Konsument(LPVOID)
{
        while(1)
        {
                Sleep((rand()%10)*150 + 100);
                WaitForSingleObject(theElementy,INFINITE);
                bufor[wy_indeks] = 0;
                wy_indeks = (wy_indeks + 1) % N;
                ReleaseSemaphore(theMiejsca,1,NULL);
                RedrawWindow(hwndApp,NULL,NULL,RDW_INTERNALPAINT);
        };
        return 0;
};

//---------------------------------------------------------------------

void DrawWindow()
{
        HDC theDC = GetDC(hwndApp);
        HBRUSH theWhiteBrush    = CreateSolidBrush(RGB(255,255,255));
        HBRUSH theBlueBrush     = CreateSolidBrush(RGB(50,50,150));
        SetTextColor(theDC,RGB(0,0,0));
        SetBkMode(theDC,TRANSPARENT);
        TextOut(theDC,135,20,"Bufor:",6);
        TextOut(theDC,45,90,"Problem producenta-konsumenta.",30);
        for(int i = 0;i < N;i++)
        {
                RECT theRect;
                theRect.left    = 9 + i*(300 / N);
                theRect.right   = 7 + (i + 1)*(300 / N);
                theRect.top     = 50;
                theRect.bottom  = 65;
                if (bufor[i])
                        FillRect(theDC,&theRect,theBlueBrush);
                else
                        FillRect(theDC,&theRect,theWhiteBrush);
        };
        DeleteObject((HBRUSH)theWhiteBrush);
        DeleteObject((HBRUSH)theBlueBrush);
        ReleaseDC(hwndApp,theDC);
};

LONG CALLBACK AppWndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam)
{
  switch (msg)
  {
       case WM_KILLFOCUS:
              break;
       case WM_SETFOCUS:
              break;
        case WM_CLOSE:
              PostQuitMessage(0);
             break;
       case WM_PAINT:
               DrawWindow();
              break;
       case WM_DESTROY:
                CloseHandle(theElementy);
                CloseHandle(theMiejsca);
                TerminateThread(theProducent,0);
                TerminateThread(theKonsument,0);
                CloseHandle(theProducent);
                CloseHandle(theKonsument);
                if (bufor) delete[] bufor;
              break;
  };
  return DefWindowProc(hwnd,msg,wParam,lParam);
}

BOOL AppInit(HINSTANCE hInst,HINSTANCE hPrev,int sw,LPSTR szCmdLine)
{
  // Rejestracja klasy okna.
  WNDCLASS cls;
  hInstApp = hInst;
  if (!hPrev)
  {
    cls.hCursor        = LoadCursor(0,IDC_ARROW);
    cls.hIcon          = NULL;
```

```
    cls.lpszMenuName   = NULL;
    cls.lpszClassName  = szAppName;
    cls.hbrBackground  = (HBRUSH)GetStockObject(LTGRAY_BRUSH);
    cls.hInstance      = hInst;
    cls.style          = CS_VREDRAW | CS_HREDRAW;
    cls.lpfnWndProc    = (WNDPROC)AppWndProc;
    cls.cbClsExtra     = 0;
    cls.cbWndExtra     = 0;
    if (!RegisterClass(&cls)) return FALSE;
  }
  // Tworzenie okna aplikacji.
  RECT rek;
  GetWindowRect(GetDesktopWindow(),&rek);
  hwndApp = CreateWindowEx(
                 WS_EX_APPWINDOW,
                 szAppName,szAppName,
               WS_POPUP | WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX,
               ((rek.right - rek.left) - 320)/2,((rek.bottom - rek.top) - 160)/2,
               320,160,0,0,hInst,0);
  ShowWindow(hwndApp,SW_SHOW);
  // Tworzymy bufor.
  bufor = new char[N];
  for(int i = 0;i < N;i++) bufor[i] = 0;
  // Tworzymy semafory.
  theElementy   = CreateSemaphore(NULL,0,N,NULL);
  theMiejsca    = CreateSemaphore(NULL,N,N,NULL);
  // Tworzymy watki.
  DWORD ID;
  theProducent  = CreateThread(NULL,0,Producent,0,0,&ID);
  theKonsument  = CreateThread(NULL,0,Konsument,0,0,&ID);
  // Inicjacja generatora losowego.
  time_t t;
  srand((unsigned) time(&t));
  return TRUE;
}

int PASCAL WinMain(HINSTANCE hInst, HINSTANCE hPrev, LPSTR szCmdLine, int sw)
{
  MSG     msg;
  if (!AppInit(hInst,hPrev,sw,szCmdLine)) return FALSE;
  for (;;)
  {
    if (PeekMessage(&msg, 0, 0, 0, PM_REMOVE))
    {
      if (msg.message == WM_QUIT)
          {
                break;
          }
      else
      {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
      };
    }
    else
    {
       WaitMessage();
    }
  }
  return msg.wParam;
}
```

Literatura.

Dodatkowe informacje można znaleźć w Win32 SDK.