Synchronizacja wątków

1 Procesy, wątki

Wątek:

- Jednostka dla której system przydziela czas procesora,
- Kontekst wątku składa się z: licznika rozkazów, stanu rejestrów, stosu (ale nie sterty!),
- Każdy proces ma co najmniej jeden wątek

Związek pomiędzy procesami a wątkami:

- Proces nie wykonuje kodu, proces jest obiektem dostarczającym wątkowi przestrzeni adresowej,
- Kod zawarty w przestrzeni adresowej procesu jest wykonywany przez wątek,
- Pierwszy wątek procesu tworzony jest implicite przez system operacyjny, każdy następny musi być utworzony explicite,
- Wszystkie wątki tego samego procesu dzielą wirtualną przestrzeń adresową i mają dostęp do tych samych zmiennych globalnych i zasobów systemowych.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

DWORD WINAPI ThreadProc(LPVOID* theArg);

int main(int argc, char *argv[])
{
  DWORD threadID;
  DWORD thread_arg = 4;

  HANDLE hThread = CreateThread( NULL, 0, (LPTHREAD_START_ROUTINE)ThreadProc,
&thread_arg, 00, &threadID);

  WaitForSingleObject( hThread, INFINITE);

  return 0;
}

DWORD ThreadProc(LPVOID* theArg)
{
    DWORD timestoprint = (DWORD)*theArg;
    for (int i = 0; i<timestoprint; i++)
      printf("Witam %d\n", i);
    return TRUE;
}
```

- 1. Zapoznaj się z opisem wszystkich funkcji systemowych użytych w tym programie.
- 2. Przeczytaj opisy funkcji WaitForSingleObject() oraz WaitForMultipleObjects() i odpowiedz na pytanie: jakie są między nimi różnice?

- 3. Zmodyfikuj przykładowy program tak, aby tworzył 4 wątki. Każdy wątek ma mieć inną nazwę przekazaną jako parametr i każdy z nich ma wypisać tę nazwę określoną ilość razy.

Nazwa wątku	Ile razy wypisać nazwę
Jurek	2
Ogórek	4
Kiełbasa	7
Sznurek	3

- Program główny ma czekać na zakończenie wszystkich wątków, a na koniec ma wypisać komunikat: "Zakończono wszystkie wątki.". Z której funkcji skorzystasz: WaitForSingleObject() czy WaitForMultipleObjects()?
- 4. Zaimplementuj funkcje omawiane na zajęciach, przedstawione jako zadanie 6.3 z książki S-G (treść sekcji krytycznej wymyśl, niech to będzie np. dostęp do pliku lub jakiejś ważnej zmiennej). Okraś kod komunikatami diagnostycznymi (np. "wątek 1 - czekam"), aby zobaczyć co naprawdę dzieje się w kodzie.

2 Synchronizacja

Win32API udostępnia 5 sposoby synchronizacji wątków. Są to:

- zdarzenia,
- mutexy,
- semafory,
- sekcje krytyczne,
- zegary oczekujące.

Mechanizm sekcji krytycznej możliwy jest do wykorzystania tylko w obrębie jednego procesu (do synchronizacji wątków), jednak jest to metoda najszybsza i najwydajniejsza. Pozostałe metody mogą być stosowane również dla wielu procesów.

2.1 Zdarzenia

Win32API umożliwia definiowanie własnych zdarzeń za pomocą funkcji CreateEvent(). Zdarzenie może być zgłoszone i obowiązuje w systemie dopóty nie nastąpi jego odwołanie. Każdy oczekujący wątek widzi więc zdarzenie jako pewną dwustanową flagę: zdarzenie jest zgłoszone albo odwołane. Za pomocą funkcji SetEvent() informujemy system o zaistnieniu zdarzenia. Od tej pory zdarzenie jest zgłoszone i wszystkie wątki oczekujące do tej pory na jego zgłoszenie mogą wznowić działanie. Zdarzenie zostaje odwołane, kiedy zostanie wywołana funkcja ResetEvent(). Na zaistnienie wydarzenia w systemie wątki oczekują za pomocą funkcji WaitForSingleObject().

Zdarzenie utworzone z ustawioną flagą ręcznego odwoływania (CreateEvent(...,TRUE,...,...)) wymaga odwołania explicite (przez ResetEvent()), natomiast zdarzenie utworzone z flagą automatycznego odwoływania (CreateEvent(...,FALSE,...,...)) zostaje odwołane automatycznie po przepuszczeniu jednego wątku przez funkcję oczekującą.

Warto również omówić działanie funkcji PulseEvent(). Otóż powoduje ona zgłoszenie zdarzenia, po czym natychmiastowe jego odwołanie. Działanie oczekujących wątków zależy od tego, czy zdarzenie jest odwoływane automatycznie czy ręcznie (patrz paragraf wyżej): jeśli zdarzenie odwoływane jest ręcznie, to funkcja PulseEvent() przepuszcza wszystkie wątki oczekujące w danej chwili na zdarzenie, po czym odwołuje zdarzenie, jeśli zaś zdarzenie odwoływane jest automatycznie, to funkcja PulseEvent() przepuszcza tylko jeden wątek z puli oczekujących w danej chwili wątków, po czym odwołuje zdarzenie.

```
void main(void)
{
      HANDLE hThread[2];
      DWORD threadID1, threadID2;
      char szFileName=”c:\\myfolder\\myfile.txt”;

      hEvent=CreateEvent(NULL, TRUE, FALSE, “FILE_EXISTS”);

          // tworzymy dwa wątki które czekają na utworzenie pliku
      hThread[0]=CreateThread(NULL, 0, ThreadProc1, szFileName, 0,
&threadID1);
      hThread[1]=CreateThread(NULL, 0, ThreadProc2, szFileName, 0,
&threadID1);

      HANDLE hFile=CreateFile(szFileName, GENERIC_WRITE, 0, &security, . .
.);

          // kod wypełniający plik danymi np. WriteFile(...)



      // sygnalizacja wątkom tego, że dane są gotowe
      // wątki od ich utworzenia tylko na to czekały
      SetEvent(hEvent);
      WaitForMultipleObjects(2, hThread, TRUE, _czas_czekania_);

      CloseHandle(hEvent);
      CloseHandle(hFile);
      CloseHandle(hThread[0]);
      CloseHandle(hThread[1]);
}


DWORD ThreadProc1(LPVOID* arg)
{
      char szFileName = (char*)arg;
// tutaj wątek otwiera zdarzenie określone w module głównym
      HANDLE hEvent = OpenEvent(SYNCHRONIZE, FALSE, “FILE_EXISTS”);
// czeka na jego pojawienie się
      WaitForSingleObject(hEvent, INFINITE);
// i czyta dane zapisane do pliku
      HANDLE hAnswerFile = ::CreateFile(szFileName, GENERIC_READ, 0, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

// przetwarzaj dane
      return TRUE;
}
```

```
DWORD ThreadProc2(LPVOID* arg)
{
      char szFileName = (char*)arg;
// tutaj wątek otwiera zdarzenie określone w module głównym
      HANDLE hEvent = OpenEvent(SYNCHRONIZE, FALSE, "FILE_EXISTS");
// czeka na jego pojawienie się
      WaitForSingleObject(hEvent, INFINITE);
// i czyta dane zapisane do pliku
      HANDLE hAnswerFile = ::CreateFile(szFileName, GENERIC_READ, 0, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

// przetwarzaj dane
      return TRUE;
}
```

Zadania:

- 5. Uzupełnij luki w kodzie tak, aby moduł główny rzeczywiście zapisywał do pliku jakieś dane (np. krótki tekst, a wątki żeby czekały na ten zapis, odczytywały tekst i wyprowadzały do na konsolę).
- 6. Czym różnią się znane z biblioteki standardowej języka C funkcje do operacji na plikach (fopen(), fread(), fclose()) od tych udostępnianych przez Win32API? Naucz się korzystać z nowych funkcji.
- 7. Skoro Win32API udostępnia swoje wersje funkcji znanych z biblioteki standardowej języka C, to może udostępnia także swoją wersję funkcji printf()? Znajdź ją i naucz się z niej korzystać.

2.2 Mutexy

Nazwa mutex pochodzi od angielskiego terminu mutually exclusive (wzajemnie wykluczający się). Mutex jest obiektem służącym do synchronizacji. Jego stan jest ustawiony jako 'sygnalizowany', kiedy żaden wątek nie sprawuje nad nim kontroli oraz 'niesygnalizowany' kiedy jakiś wątek sprawuje nad nim kontrolę. Synchronizację za pomocą mutexów realizuje się tak, że każdy wątek czeka na objęcie mutexa w posiadanie, zaś po zakończeniu operacji wymagającej wyłączności, wątek uwalnia mutexa.

W celu stworzenia mutexa, wątek woła funkcję CreateMutex(). W chwili tworzenia wątek może zażądać natychmiastowego prawa własności do mutexa. Inne wątki (nawet innych procesów) otwierają mutexa za pomocą funkcji OpenMutex(). Następnie czekają na objęcie mutexa w posiadanie. Do uwalniania mutexów służy funkcja ReleaseMutex().

Jeśli wątek kończy się bez uwalniania mutexów, które posiadał, takie mutexy uważa się za porzucone. Każdy czekający wątek może objąć takie mutexy w posiadanie, zaś funkcja czekająca na przydział mutexa (WaitForSingleObject(), jak widać bardzo uniwersalna funkcja) zwraca wartość WAIT_ABANDONED. W takiej sytuacji warto zastanowić się, czy gdzieś nie wystąpił jakiś błąd (skoro wątek, który był w posiadaniu mutexa nie oddał go explicite przez ReleaseMutex(), to najprawdopodobniej został zakończony w jakiś nieprzewidziany sposób). Mutexy są w działaniu bardzo podobne do semaforów. O różnicach między nimi proszę przeczytać przy opisie semaforów.

```
void main(void)
{
```

```
      HANDLE hThread[2];
      DWORD threadID1, threadID2;

      char szFileName=”c:\\myfolder\\myfile.txt”;

      hMutex=CreateMutex(NULL, TRUE, “FILE_EXISTS”);

// tworzymy dwa wątki które czekają na utworzenie pliku

      hThread[0]=CreateThread(NULL, 0, ThreadProc1, &hMutex, 0, &threadID1);

      hThread[1]=CreateThread(NULL, 0, ThreadProc2, &hMutex, 0, &threadID1);

      HANDLE hFile=CreateFile(szFileName, GENERIC_WRITE, 0, &security, . .
.);

// kod wypełniający plik danymi np. WriteFile(...)


// sygnalizacja wątkom tego, że dane są gotowe
// wątki od ich utworzenia tylko na to czekały

      ReleaseMutex(hMutex);
      WaitForMultipleObjects(2, hThread, TRUE, _czas_czekania_);

      CloseHandle(hMutex);
      CloseHandle(hFile);
      CloseHandle(hThread[0]);
      CloseHandle(hThread[1]);
}

DWORD ThreadProc1(LPVOID* arg)
{
      HANDLE hMutex = (HANDLE)(*arg);
      WaitForSingleObject(hMutex, INFINITE);

// i czyta dane zapisane do pliku
      HANDLE hAnswerFile = ::CreateFile(szFileName, GENERIC_READ, 0, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

// przetwarzaj dane
      ReleaseMutex(hMutex);
      return TRUE;
}

DWORD ThreadProc2(LPVOID* arg)
{
      HANDLE hMutex = (HANDLE)(*arg);
      WaitForSingleObject(hMutex, INFINITE);

// i czyta dane zapisane do pliku
      HANDLE hAnswerFile = ::CreateFile(szFileName, GENERIC_READ, 0, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

// przetwarzaj dane
      ReleaseMutex(hMutex);
      return TRUE;
```

```
}
```

- 8. Uzupełnij luki w kodzie tak, aby moduł główny rzeczywiście zapisywał do pliku jakieś dane (np. krótki tekst, a wątki żeby czekały na ten zapis, odczytywały tekst i wyprowadzały do na konsolę).

2.3 Semafory

Semafory mogą być wykorzystywane tam, gdzie zasób dzielony jest na ograniczoną ilość użytkowników. Semafor działa jak furtka kontrolująca ilość wątków wykonujących jakiś fragment kodu. Za pomocą semaforów aplikacja może kontrolować na przykład maksymalną ilość otwartych plików, czy utworzonych okien. Semafory są w działaniu bardzo podobne do mutexów.

Nowy semafor tworzony jest w funkcji CreateSemaphore(). Wątek tworzący semafor specyfikuje wartość wstępną i maksymalną licznika. Inne wątki uzyskują dostęp do semafora za pomocą funkcji OpenSemaphore() i czekają na wejście za pomocą funkcji ... (to już powinno być jasne jakiej).

Po zakończeniu pracy w sekcji krytycznej wątek uwalnia semafor za pomocą funkcji ReleaseSemaphore().

Wątki nie wchodzą w posiadanie semaforów! W przypadku mutexów, jeśli wątek zażąda po raz kolejny dostępu do tego mutexu, którego jest już właścicielem, dostęp taki zostaje mu przyznany natychmiast. Jeśli wątek nagle rozpocznie czekanie na ten sam semafor, to semafor zachowuje się tak, jakby wejścia zażądał każdy inny wątek. Inaczej wygląda także sprawa uwalniania semaforów i mutexów: mutex może być uwolniony tylko przez wątek, który jest jego właścicielem, licznik semafora może być zwiększony przez dowolny wątek, który z tego semafora korzysta.

```
void main(void)
{
      HANDLE hThread[2];
      DWORD threadID1, threadID2;
      char szFileName=”c:\\myfolder\\myfile.txt”;

      hSemaphore=CreateSemaphore(NULL, 0, 1, “FILE_EXISTS”);

// tworzymy dwa wątki które czekają na utworzenie pliku
      hThread[0]=CreateThread(NULL, 0, ThreadProc1, &hSemaphore, 0,
&threadID1);

      hThread[1]=CreateThread(NULL, 0, ThreadProc2, &hSemaphore, 0,
&threadID1);

      HANDLE hFile=CreateFile(szFileName, GENERIC_WRITE, 0, &security, . .
.);

// kod wypełniający plik danymi np. WriteFile(...)

// sygnalizacja wątkom tego, że dane są gotowe
// wątki od ich utworzenia tylko na to czekały

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

```
	WaitForMultipleObjects(2, hThread, TRUE, _czas_czekania_);

	CloseHandle(hSemaphore);
	CloseHandle(hFile);
	CloseHandle(hThread[0]);
	CloseHandle(hThread[1]);
}

DWORD ThreadProc1(LPVOID* arg)
{
	HANDLE hSem = OpenSemaphore( SEMAPHORE_ALL_ACCESS, "FILE_EXISTS");
	WaitForSingleObject(hSem, INFINITE);

// i czyta dane zapisane do pliku

	HANDLE hAnswerFile = ::CreateFile(szFileName, GENERIC_READ, 0, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

// przetwarzaj dane

	ReleaseSemaphore(hSem, 1, NULL);
	return TRUE;
}

DWORD ThreadProc2(LPVOID* arg)
{
	HANDLE hSem = OpenSemaphore( SEMAPHORE_ALL_ACCESS, "FILE_EXISTS");
	WaitForSingleObject(hSem, INFINITE);

// i czyta dane zapisane do pliku

	HANDLE hAnswerFile = ::CreateFile(szFileName, GENERIC_READ, 0, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

// przetwarzaj dane

	ReleaseSemaphore(hSem, 1, NULL);
	return TRUE;
}
```

- 9. Uzupełnij luki w kodzie tak, aby moduł główny rzeczywiście zapisywał do pliku jakieś dane (np. krótki tekst, a wątki żeby czekały na ten zapis, odczytywały tekst i wyprowadzały do na konsolę).
- 10. Omów różnice między semaforami i mutexami. Czasami obiektów służących do synchronizacji można zupełnie nieoczekiwanie użyć do rozwiązania innych problemów. Zastanów się jak wykorzystać mutexy do następującego: w jaki sposób wykryć obecność aplikacji w systemie, aby zabezpieczyć się przed kilkukrotnym uruchamianiem jej jednocześnie. Napisz odpowiedni program. Program powinien uruchomić się w oknie konsoli i czekać na naciśnięcie dowolnego klawisza, po czym zakończyć działanie. Próba uruchomienia drugiej kopii programu powinna zakończyć się komunikatem "Program jest już uruchomiony" po czym natychmiast zakończyć działanie.

2.4 Sekcja krytyczna

Interfejs programowania WinAPI udostępnia typ danych CRITICAL_SECTION, który wraz z odpowiednim zestawem funkcji może być wykorzystany do implementacji sekcji krytycznej.

Prototypy funkcji:

VOID InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
VOID EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
VOID LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
VOID DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection);

BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection); // tylko WinNT!

Przykład programu z sekcją krytyczną.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define MAXTRY 3

CRITICAL_SECTION cs;          // dzielona na wszystkie wątki

// główny wątek programu
void ThreadMain(char *name)
{
      int i;

      for (i=0; i<MAXTRY; i++)
      {
            EnterCriticalSection(&cs);

            /* proszę spróbować też zamiast powyższej linii napisać

                        while ( TryEnterCriticalSection(&cs)==FALSE )
              {
                  printf("%s, czekam na wejście\n", name);
                   Sleep(5);
              }

                        uwaga! - tylko na WinNT
            */

            printf("%s, jestem w sekcji krytycznej!\n", name);
            Sleep(5);
            LeaveCritcalSection(&cs);

            printf("%s, wyszedłem z sekcji krytycznej!\n", name);
      }

}

// tworzy wątek potomny

HANDLE CreateChild(char* name)
{
      HANDLE hThread; DWORD dwId;
```

```
      hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadMain,
(LPVOID)name, 0, &dwId);

      assert(hThread!=NULL); return hThread;
}

int main(void)
{
      HANDLE hT[4];

      InitializeCriticalSection(&cs);

      hT[0]=CreateChild("Jurek");
      hT[1]=CreateChild("Ogórek");
      hT[2]=CreateChild("Kiełbasa");
      hT[3]=CreateChild("Sznurek");

      WaitForMultipleObjects(4, hT, TRUE, INFINITE);

      CloseHandle(hT[0]);CloseHandle(hT[1]);
      CloseHandle(hT[2]);CloseHandle(hT[3]);

      DeleteCriticalSection(&cs);

      return 0;
}
```