Laboratorium Architektury Komputerów
i Systemów Operacyjnych

Ćwiczenie 5

Synchronizacja procesów i wątków

Procesy a wątki

Wątki wykonują się w jednej przestrzeni adresowej i w odróżnieniu od zwykłych procesów nie są niezależne od siebie. Interakcja pomiędzy wątkami tego samego procesu nie wymaga stosowania żadnych specjalnych mechanizmów, ponad zwykłe wykonywanie programu i dostęp do pamięci. Wszystkie wątki działające w ramach jednego procesu mają natychmiastowy dostęp do wszystkich zmiennych globalnych zdefiniowanych w programie.

Niesie to potencjalne zagrożenie niewłaściwego korzystania ze współdzielonych zasobów. Metodami eliminującymi powyższe zagrożenie są mechanizmy synchronizacji i maksymalna separacja obszarów pamięci używanych przez wątki. W przypadku odrębnych procesów, separacja była gwarantowana przez system operacyjny, w przypadku wątków zaś — należy o nią zadbać poprzez starannie zaprojektowanie kodu.

Dostępne mechanizmy synchronizacji są bardzo podobnie jak w przypadku procesów (semafory, komunikaty, sekcje krytyczne itd.) i służą do ustalenia jednego wątku, który może korzystać z określonego zasobu współdzielonego. We współpracy międzyprocesowej, wykorzystywane były mechanizmy *nazwane*, wymagające podania opisowego identyfikatora, dzięki któremu system operacyjny mógł scalić ze sobą ich wystąpienia w różnych procesach w pojedynczy, współdzielony byt. W przypadku wątków wewnątrz jednego procesu, wykorzystuje się zaś elementy *nienazwane*, ponieważ wszystko w ramach procesu już jest współdzielone z definicji, a rozróżniania ich wystąpień dokonuje sam programista w tworzonym przez siebie kodzie, poprzez użycie właściwych zmiennych.

Jedną z najważniejszych różnic między tworzeniem kodu dla wielu procesów i dla wielu wątków, jest jego reakcja na potencjalne błędy programisty. W pierwszym przypadku wystąpienie krytycznego błędu ma efekt ograniczony – jeden proces ulega zniszczeniu, inne mogą kontynuować (o ile są w stanie bez niego). W przypadku programowania wielowątkowego, krytyczny błąd w jednym wątku ma szansę tak jak poprzednio, zniszczyć cały proces – czyli wszystkie wątki. Z tego powodu, rozważając programowanie wykorzystujące elementy równoległości, należy zawsze rozważyć, czy w danym przypadku lepiej jest zastosować bezpieczniejszy podział na procesy, czy też podział na wątki dający względną prostotę kodu i potencjalnie szybszą komunikację.

Problemami wydajnego współdzielenia zasobów oraz projektowania bezpiecznych algorytmów wykonywanych równolegle zajmuje się dziedzina *przetwarzanie współbieżne i rozproszone*.

Powstawanie wątków

Utworzenie procesu oznacza automatyczne utworzenie jednego wątku — tzw. wątku głównego (wątku inicjalnego). W praktyce, wątek jest jedyną jednostką wykonywalną dla procesu. Proces istnieje tak długo, dopóki zawiera jakieś wątki. Utworzenie wątku głównego jest niewidoczne dla programisty.

Po wykonaniu funkcji tworzącej nowy wątek, funkcja wątku może być wykonywana natychmiast w sposób współbieżny z już istniejącymi wątkami. Wątek może być utworzony także jako *wstrzymany* — zostanie on przygotowany przez system operacyjny i uruchomienie go będzie możliwe w dowolnym, późniejszym momencie.

System przydziela każdemu aktywnemu wątkowi określony czas procesora (powstaje iluzja jednoczesnego wykonywania wątków — w systemach wieloprocesorowych wątki wykonywane są rzeczywiście jednocześnie). W systemie Windows przyjęto planowanie realizacji procesów i wątków na bazie *priorytetów wątków*. Do realizacji przekazywany jest wątek (gotowy do wykonania) o najwyższym priorytecie. Wątek jest wykonywany przez czas nazywany *kwantem* (typowa wartość 10 ms). Kwant określa jak długo będzie wykonywany wątek, dopóki system operacyjny nie odbierze mu procesora. Po upływie kwantu czasu system operacyjny sprawdza czy są inne gotowe wątki z takim samym priorytetem lub wyższym. Jeśli takich wątków nie ma, to bieżący wątek otrzymuje jeszcze jeden kwant. Jednak wątek może nie wykorzystać w pełni swojego kwantu. Jeśli tylko inny wątek z wyższym priorytetem jest gotowy do wykonania, to bieżący wątek zostaje wstrzymany, także w przypadku jeśli kwant nie został jeszcze wyczerpany.

W każdym przypadku system operacyjny wybiera następny wątek do wykonania. Wybranie nowego (innego) wątku wymaga przełączenia kontekstu, co wykonuje system operacyjny (zapamiętanie rejestrów, wskaźników obu stosów, wskaźnik przestrzeni adresowej). Takie same parametry są odczytywane dla uruchamianego wątku.

Funkcje operujące na procesach i wątkach dostępne są zarówno w systemie MS Windows jak i w systemie Linux. Funkcje dostępne w Linuxie są zgodne ze specyfikacją POSIX, zaś w systemie Windows zdefiniowane są w interfejsie WinAPI. Nazwy i parametry funkcji używanych w obu specyfikacjach (POSIX/WinAPI) różnią się dość znacznie, ale podstawowe reguły tworzenia programów wielowątkowych są bardzo podobne. W dalszej części niniejszego opracowania skupimy uwagę na funkcjach dostępnych w systemie Windows.

Funkcje zdefiniowane w WinAPI często wymagają przekazywania jako argument pewnej struktury, nazywanej *dojściem* lub *uchwytem*. Uchwyt jest to zmienną typu HANDLE — jest ona w istocie wskaźnikiem, więc jest widoczna tylko w ramach przestrzeni adresowej jednego procesu. Uchwyty zazwyczaj otrzymuje się w wyniku wykonania funkcji *Create...* lub *Open...*. Ilość możliwych do utworzenia uchwytów jest ograniczona. Po zakończeniu korzystania z nich należy je bezwzględnie zwolnić za pomocą funkcji *CloseHandle*, w innym przypadku mogą się wyczerpać i nie będzie już możliwe tworzenie nowych.

Wielu elementom dostępnym przez uchwyty, w tym procesom i wątkom, towarzyszy także obok uchwytu inna wartość, nazywana *identyfikatorem* — jest to numer jednoznacznie określający dany obiekt w systemie. Identyfikator, w przeciwieństwie do uchwytu, może być swobodnie przekazywany między procesami (np. za pomocą mechanizmu komunikatów). Z identyfikatorów nie można jednak korzystać bezpośrednio, większość funkcji zdefiniowanych

w WinAPI wymaga podawania uchwytów! Do przekształcenia identyfikatora w uchwyt służą wspomniane już funkcje *Open.*.

W systemie MS Windows do tworzenia wątków używana jest funkcja CreateThread, zdefiniowana w WinAPI. Zaleca się jednak, o ile jest to tylko możliwe, używanie POSIX'owej funkcji _beginthreadex, na wypadek gdyby program trzeba dostosować do pracy na innym systemie operacyjnym. Obydwie funkcje wymagają, aby poprzez parametry podać im dwie ważne informacje:

- funkcję wątku,
- parametr startowy wątku.

Pierwszy parametr "funkcja wątku" jest niczym innym jak konkretną funkcją, która zostanie wykonana przez wątek. Można to rozumieć, jako określenie funkcji *main* dla nowego wątku. Parametr startowy jest zaś opcjonalną, *pojedynczą* wartością, która ma być temu konkretnemu wątkowi przekazana na starcie. Z racji ograniczenia do jednej wartości, zazwyczaj parametr startowy jest wskaźnikiem do struktury, tak aby można było przekazać poprzez niego dowolnie dużą strukturę zawierającą dane potrzebne wątkowi do pracy.

Wątki a funkcje systemowe

Poszczególne wątki mogą wywoływać funkcje systemowe. W szczególności pewien wątek może wywołać funkcję, która wcześniej została wywołana przez inny wątek i dotychczas nie została zakończona. Oznacza to, że kod i dane funkcji powinny być tak skonstruowane, że możliwe są kolejne wywołanie funkcji przed zakończeniem realizacji poprzednich. Funkcje posiadające taką własność noszą nazwę *funkcji wielowykonywalnych* (ang. reentrant). Praktycznie, funkcja jest wielowykonywalna, jeśli korzysta wyłącznie z danych i wyników pośrednich umieszczonych na stosie. Jeśli funkcja wykonuje działania na danych statycznych (deklarowanych z użyciem kwalifikatora static), to funkcja ta zazwyczaj nie jest wielowykonywalna.

W oprogramowaniu Microsoft Visual Studio dostępnych jest kilka bibliotek funkcji języka C, dostosowanych do różnych aplikacji. Biblioteka LIBC.LIB używana jest przy konsolidacji (linkowaniu) statycznej programów jednowątkowych, natomiast biblioteka LIBCMT.LIB używana jest dla programów wielowątkowych. Wybór pomiędzy nimi konfiguruje się na poziomie projektu. W jego właściwościach, w grupie *C/C++* w *Code Generation* znajduje się opcja *Runtime Library* pozwalająca na przełączenie linkowania pomiędzy biblioteką do pracy jedno- lub wielowątkowej. W nowszych wersjach środowiska domyślnie wybrana jest biblioteka dla programów wielowątkowych.

Funkcje oczekiwania

Funkcja Sleep powoduje wejście wątku w stan uśpienia na zadaną liczbę milisekund. Wywołanie z argumentem równym zero powoduje, że wątek rezygnuje z pozostałego kwantu czasu na rzecz innego wątku o tym samym priorytecie, który jest gotowy do wykonania. Jeśli takich wątków nie ma, to wykonywanie funkcji Sleep zostaje zakończone i kontynuowane jest wykonywanie wątku.

W technice synchronizacji wątków kluczowe znaczenie ma funkcja WaitForSingleObject(). Funkcja ta zawiesza wykonanie wątku aż do chwili, gdy stan testowanego obiektu przyjmie określoną wartość. Ściślej, jeśli obiekt jest w stanie *niesygnalizowany*, to wątek wywołujący wchodzi w stan oczekiwania. W tym stanie wątek

pochłania bardzo mało czasu procesora oczekując, aż stan obiektu będzie *sygnalizowany* lub upłynął określony czas. Przed powrotem funkcja WaitForSingleObject() modyfikuje stan synchronizowanych obiektów, np. zmniejsza licznik obiektu semafora o 1, wprowadza go w stan *niesygnalizowany*, itp..

Ponieważ funkcja ta stosowna jest do obiektów różnego typu, więc będziemy do niej powracać przy omawianiu tych obiektów. Pierwszym argumentem funkcji jest *uchwyt* do obiektu. Drugim argumentem funkcji WaitForSingleObject() jest maksymalny czas oczekiwania. Jeśli stan obiektu nie zmieni się w zadanym czasie, to funkcja zwraca wartość WAIT_TIMEOUT, co pozwala odpowiednio pokierować dalszym wykonywaniem wątku.

Jeśli oczekujemy na zmianę stanu kilku obiektów, to w takim przypadku należy zastosować funkcję WaitForMultipleObjects(). Można oczekiwać na zmianę stanu wszystkich obiektów, albo tylko jednego z nich.

Program przykładowy

Poniżej podano przykład prostego programu, w którym tworzone są dwa wątki. Oba wątki wykonują tę samą funkcję wątku, wewnątrz której (za pomocą *MessageBox*) wyświetlany jest komunikat na ekranie.

Kod funkcji wątku `FunkcjaWatku` podany jest w początkowej części programu. Zauważmy, że funkcja ma tylko jeden argument będący wskaźnikiem do wartości typu `void`. Poprzez ten argument do funkcji wątku przekazywany jest adres struktury, wewnątrz której umieszczone są właściwe argumenty funkcji. Zatem w początkowej części programu trzeba zdefiniować strukturę, w której zostaną określone wszystkie argumenty funkcji wątku — wskaźnik do tej struktury jest argumentem funkcji wątku (temat ten będzie szczegółowo omawiany w dalszej części).

W podanym tu przykładzie do funkcji wątku przekazywany jest adres tablicy zawierającej krótki tekst informacyjny `Pierwszy watek` lub `Drugi watek`. Ponieważ przekazywany jest tylko jeden argument, nie ma potrzeby tworzenia struktury, a przekazany do funkcji wskaźnik typu `void *` jest od razu rzutowany na wskaźnik `char *`. Wyznaczony wskaźnik jest drugim argumentem funkcji `MessageBox`, która wyświetla komunikat na ekranie.

W końcowej części funkcji wątku wywoływana jest funkcja `_endthreadex`, która kończy wykonywanie funkcji wątku i likwiduje wątek.

Wykonywanie omawianego programu rozpoczyna się od kodu zawartego wewnątrz funkcji `main()`. Po zdefiniowaniu tablic `w1` i `w2`, tworzone są dwa nowe wątki za pomocą funkcji `_beginthreadex`. Funkcja ta ma 6 parametrów, spośród których najważniejszy są trzeci, czwarty i szósty. Trzeci parametr podaje wskaźnik do funkcji wątku, która jest wykonywana w ramach utworzonego wątku. Parametr czwarty jest wskaźnikiem do struktury zawierającej argumenty funkcji wątku (zob. opis wyżej), a parametr szósty jest wskaźnikiem do elementu tablicy, w którym zostanie zapisany identyfikator utworzonego wątku.

Po utworzeniu nowego wątku rozpoczyna się wykonywanie funkcji wątku, skojarzonej z tym wątkiem. Opisana wyżej funkcja wątku `FunkcjaWatku` wyświetla tylko komunikat na ekranie i kończy swoje działanie.

Po utworzeniu dwóch nowych wątków, wykonywany jest dalej kod zawarty w wątku głównym. Funkcja `WaitForMultipleObjects(2, hWatki, TRUE, INFINITE)` wstrzymuje działanie wątku głównego aż do zakończenia realizacji dwóch niedawno utworzonych wątków. Funkcja ta ma cztery argumenty, z których drugi opisuje tablicę zawierającą uchwyty do utworzonych wątków, a pierwszy argument podaje rozmiar tej

tablicy. Trzeci argument (tu: `TRUE`) opisuje warunek zakończenia oczekiwania: wartość `TRUE` oznacza, że funkcja zakończy swoje działanie dopiero zakończeniu działania obu wątków. Czwarty parametr o wartości `INFINITE` oznacza, że czas oczekiwania nie jest ograniczony.

Zatem, wątek główny wznowi swoje działanie dopiero po zakończeniu wykonywania obu utworzonych wątków. Wówczas wyświetlany jest komunikat `Wykonywanie obu wątków zostało zakończone` i wykonywanie całego programu zostaje zakończone.

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

/*-------------------------------------------------------*/

unsigned __stdcall FunkcjaWatku(void * arg) // funkcja wątku
{
     char * znaki = (char *) arg;
     MessageBox(0, znaki, "Testowanie wątków",0);
     _endthreadex(0);
     return 0;
};

/*-------------------------------------------------------*/

int main()
{
     unsigned IdentWatku[2];
     HANDLE hWatki[2];

     char  w1[] = "Pierwszy wątek"; // parametr dla watku nr 1
     char  w2[] = "Drugi wątek";   //parametr dla watku nr 2

     hWatki[0] = (HANDLE)_beginthreadex (NULL, 0,
               &FunkcjaWatku, (void *)w1, 0, &IdentWatku[0]);
     hWatki[1] = (HANDLE)_beginthreadex (NULL, 0,
               &FunkcjaWatku, (void *)w2, 0, &IdentWatku[1]);

     // oczekiwanie na zakończenie wykonywania wątków
     WaitForMultipleObjects(2, hWatki, TRUE, INFINITE);

     // zamknięcie uchwytów (handle) wątków
     CloseHandle(hWatki[0]);
     CloseHandle(hWatki[1]);

     MessageBox(0,
               "Wykonywanie obu wątków zostało zakończone",
               "Testowanie wątków",0);

     return 0;
}
```

Synchronizacja wątków i procesów

Jednym z podstawowych zagadnień przy przetwarzaniu wielowątkowym jest zapewnienie odpowiedniej kolejności operacji wykonywanych przez wątki, a także zapewnienie wyłącznego dostępu do określonych zasobów. Praktycznie synchronizacja wątków polega na tym, że wątek wprowadza siebie w stan uśpienia. Przedtem jednak informuje system na jakie zdarzenia oczekuje, które mają wznowić jego wykonywanie. Jeśli zdarzenie wystąpi, to wątek zostaje umieszczony w kolejce wątków gotowych, oczekujących na wykonanie przez procesor (lub procesory).

System Windows oferuje kilka sposobów synchronizacji wątków — są one dość podobne, wybór głównie zależy od pewnych niuansów ich zachowania. Należą do nich m.in.: *zdarzenia* (ang. events), *mutexy (omawiane w ćwiczeniu 4)*, *semafory*, *sekcje krytyczne* i *zegary oczekujące* (ang. waitable timers). Ze wszystkich technik synchronizacji sekcja krytyczna jest najprostsza do realizacji i wykorzystania, jednak może ona synchronizować wątki wewnątrz pojedynczego procesu. Inne z wymienionych sposobów, w swoich *nazwanych* wersjach, mogą być stosowane także do synchronizacji procesów.

Sekcje krytyczne

Formalnie, termin *sekcja krytyczna* określa pewien fragment kodu programu, który z racji swoich efektów ubocznych, nie ma prawa być wykonywany współbieżnie z innymi fragmentami, gdyż wtedy natura jego konstrukcji grozi powstawaniem błędów. W systemie Windows stworzono prosty sposób pozwalający na oznaczanie takich miejsc. Początek sekcji krytycznej można wskazać wywołując funkcję `EnterCriticalSection`, a jej zakończenie – symetryczną funkcją `LeaveCriticalSection`. System operacyjny zapewni wtedy, że w danej chwili tylko jeden wątek może znajdować się wewnątrz tak oznaczonej sekcji krytycznej. Jeśli w czasie wykonywania kodu sekcji krytycznej inny wątek spróbuje wykonać funkcję `EnterCriticalSection`, to jest zatrzymywany aż do chwili, gdy bieżący wątek opuści sekcję krytyczną.

W celu wyróżnienia kilku *niezależnych* sekcji krytycznych, system operacyjny wymaga, aby obu wyżej wymienionym funkcjom podawać coś, co wskaże, do(z) której sekcji krytycznej właśnie wątek wchodzi lub wychodzi. W tym celu tworzy się, dla każdej sekcji krytycznej specjalną zmienną typu `CRITICAL_SECTION` i:

- wchodząc do sekcji krytycznej, wywołuje się funkcję `EnterCriticalSection` podając zmienną właściwą dla tej sekcji krytycznej;
- wychodząc z sekcji krytycznej, wywołuje się funkcję `LeaveCriticalSection` podając zmienną właściwą dla tej sekcji krytycznej.

Przykład programu, w którym stosowane są sekcje krytyczne podany jest dalej.

Z racji niefortunnego doboru nazwy typu `CRITICAL_SECTION`, często stosuje się skrót myślowy i tę właśnie zmienną identyfikującą nazywa się „sekcją krytyczną” i postrzega jako mechanizm tak samo jak mutex czy semafor. Faktycznym mechanizmem jest tutaj jednak wywołanie funkcji `Enter../Leave..`, a właściwa sekcja krytyczna leży pomiędzy tymi wywołaniami.

Praktycznie, w początkowej części programu, w obszarze zmiennych globalnych tworzy się zmienną identyfikującą sekcję krytyczną, np.:

```
CRITICAL_SECTION  sekcja_krytyczna;
```

W początkowej części kodu funkcji `main` trzeba zainicjalizować sekcję krytyczną, np.:

```
InitializeCriticalSection (& sekcja_krytyczna);
```

W dalszej części programu, posługując się tą zmienną można tworzyć fragmenty kodu programu mające charakter sekcji krytycznych, np.

```
EnterCriticalSection(&sekcja_krytyczna);
- - - - - - - - - - - - - - - - - - - - - -
(tu znajdują się instrukcje stanowiące
 sekcję krytyczną)
- - - - - - - - - - - - - - - - - - - - - -
LeaveCriticalSection(&sekcja_krytyczna);
```

Tak jak wcześniej powiedzieliśmy, jeśli wątek wejdzie do sekcji krytycznej, to żaden inny wątek nie będzie mógł wejść do sekcji krytycznej dopóki ten pierwszy nie opuści sekcji krytycznej (poprzez wykonanie funkcji `LeaveCriticalSection`).

Przekazywanie parametrów do funkcji wątku

Parametry przekazywane do funkcji wątku mogą być bardzo różnorodne pod względem ich liczby i typów. W tej sytuacji przyjęto rozwiązanie „uniwersalne” w postaci przekazywania tylko jednego parametru będącego wskaźnikiem do typu `void`. Wskaźnik ten, poprzez operację rzutowania, może oznaczać adres pojedynczej zmiennej, jak również adres grupy zmiennych czy tablic, zintegrowanych w formie struktury `struct`. W rezultacie, niezależnie od liczby i typów rzeczywiście przekazywanych parametrów, do funkcji wątku przekazywany jest zawsze jeden parametr w postaci wskaźnika.

W celu bliższego wyjaśnienia stosowanych technik programistycznych rozpatrzymy przykład prostego programu, w którym funkcja wątku oblicza pole prostokąta. Przekazywane parametry obejmują nazwę prostokąta, długość i szerokość. Dla wygody te trzy parametry zostały włączone do struktury, której (za pomocą deklaracji `typedef`) nadano nazwę `WYMIARY`.

```
typedef struct _wymiary
{
  char  * nazwa;
  double  dlugosc;
  double  szerokosc;
} WYMIARY;
```

W takim ujęciu symbol `WYMIARY` może być traktowany jako nowy typ danych. W początkowej części programu głównego (`main`) za pomocą deklaracji

```
WYMIARY  prostokat_A, prostokat_B;
```

tworzone są dwie struktury przeznaczone do przekazywania danych, odpowiednio, do wątków A i B. W dalszej części programu (przed utworzeniem wątków) omawiane struktury wypełniane są wartościami, np. do pola `dlugosc` w strukturze `prostokat_B` wpisywana jest wartość 3.24:

```
prostokat_B.dlugosc = 3.24;
```

Z kolei w trakcie tworzenia wątków (funkcja `_beginthreadex`) wskaźniki do tych struktur, rzutowane na typ `void *`, występują jako czwarty parametr funkcji, np.:

```
hWatki[1] = (HANDLE) _beginthreadex (NULL, 0, pole_prostokata,
          (void *) &prostokat_B, 0, &Identyfikator_watku[1]);
```

W rezultacie funkcja wątku otrzyma omawiane parametry (nazwa, długość, szerokość) w postaci wskaźnika na typ `void *`.

Wewnątrz funkcji wątku otrzymany wskaźnik `dlugosci_bokow` jest rzutowany na typ `WYMIARY *`:

```
WYMIARY * wsk = (WYMIARY *) dlugosci_bokow;
```

co umożliwia odczytywanie wartości poszczególnych pól. Przykładowo, wartość pola szerokość wyznaczana jest z pomocą poniższej instrukcji:

```
  double y = wsk -> szerokosc;
```

Uwaga: operator `->` zapisujemy w postaci pary znaków - i > (nie jest to znak strzałki występujący na klawiaturze). Operator ten wyznacza zawartość pola o podanej nazwie (tu: `szerokosc`) zawartego w strukturze wskazywanej przez wskaźnik (tu: `wsk`).
Poniżej podano pełny kod analizowanego programu.

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

// prototyp funkcji wątku
// uwaga: przed stdcall występują dwa znaki podkreślenia
unsigned  __stdcall  pole_prostokata
                            (void  *  dlugosci_bokow);


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

// definicja typu WYMIARY używanego do przekazywania
// parametrów do obu wątków
typedef struct _wymiary
{
  char  * nazwa;
  double  dlugosc;
  double  szerokosc;
} WYMIARY;

CRITICAL_SECTION  sekcja_krytyczna;

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

int main()
{
  // to jest wątek główny uruchamiany automatycznie przez
  // system

  // struktury używane do przekazywania parametrów
  // do wątków A i B
```

```
  WYMIARY  prostokat_A, prostokat_B;

  unsigned int Identyfikator_watku[2];
  HANDLE hWatki[2]; // uchwyty wątków
  char tabl_A[] = "Prostokąt A";
  char tabl_B[] = "Prostokąt B";

  printf("\nObliczanie pola prostokątów\n" );

  // wypełnianie struktur przekazywanych do wątków A i B
  prostokat_A.nazwa = tabl_A;
  prostokat_A.dlugosc = 3.0;
  prostokat_A.szerokosc = 4.0;
  prostokat_B.nazwa = tabl_B;
  prostokat_B.dlugosc = 3.24;
  prostokat_B.szerokosc = 12.75;

  // inicjalizacja zmiennej dla sekcji krytycznej
  InitializeCriticalSection (& sekcja_krytyczna);

  // tworzenie wątków A i B
  // oba wątki, w chwili uruchomienia rozpoczynają wykonywanie
  // tej samej funkcji 'pole_prostokata', lecz z różnymi
  // parametrami startowymi;
  // funkcja do wykonania przekazywana jest przez
  // parametr trzeci
  // parametry startowe wątku przekazywane są przez
  // parametr czwarty
  hWatki[0]=(HANDLE) _beginthreadex (NULL, 0, pole_prostokata,
          (void *) &prostokat_A,0,&Identyfikator_watku[0]);
  hWatki[1]=(HANDLE) _beginthreadex (NULL, 0, pole_prostokata,
          (void *)&prostokat_B, 0,&Identyfikator_watku[1]);

  // poniższy kod stanowi dalej wątek główny

  // oczekiwanie na zakończenie wykonywania wątków
  WaitForMultipleObjects(2, hWatki, TRUE, INFINITE);

  printf("\nObliczenia zostały zakończone\n");

  // zamknięcie uchwytów (handle) wątków
  CloseHandle(hWatki[0]);
  CloseHandle(hWatki[1]);

  // usunięcie sekcji krytycznej
  DeleteCriticalSection (&sekcja_krytyczna);

  return 0;
}

//---------------------------------------------------------
```

```
// funkcja wątku
unsigned  __stdcall  pole_prostokata
                                  (void  *  dlugosci_bokow)
{
  WYMIARY * wsk = (WYMIARY *) dlugosci_bokow;
  double x = wsk -> dlugosc;
  double y = wsk -> szerokosc;
  char * tekst = wsk -> nazwa;

  // pole prostokąta
  double pole = x * y;

  EnterCriticalSection(&sekcja_krytyczna);
     // wyświetlanie wyniku
     printf ("\n %s o bokach %f %f ma pole %f\n",
                                    tekst, x, y, pole);
  LeaveCriticalSection(&sekcja_krytyczna);

  _endthreadex(0);
  return 0;
};
```

Mutexy i semafory

Koncepcja mutexów omawiana były szczegółowo w poprzednim ćwiczeniu. Przypomnijmy, że termin *mutex* pochodzi od angielskiego określenia *mutually exclusive* — wzajemnie wykluczający się. Mutex może znajdować się w jednym z dwóch stanów:

- *sygnalizowany* żaden wątek nie sprawuje nad nim kontroli,
- *niesygnalizowany* jakiś wątek sprawuje kontrolę nad mutexem.

Synchronizacja za pomocą *mutexa* sprowadza się do tego, że wątki czekają na objęcie mutexa w posiadanie. Jeśli mutex jest uwolniony, to jeden z wątków obejmuje go w posiadanie, a po wykonaniu operacji, uwalnia go.

Mutex tworzy się za pomocą funkcji `CreateMutex()`. W chwili tworzenia wątek może zażądać od razu prawa własności do mutexa. Inne wątki, nawet wątki innych procesów otwierają mutex za pomocą funkcji `OpenMutex(...)`. Za pomocą funkcji `OpenMutex(...)` wątek uzyskuje uchwyt do mutexa, ale nie oznacza to objęcia mutexa w posiadanie. Mutex uwalnia się za pomocą funkcji `ReleaseMutex()`.

Praktycznie, w przypadku synchronizacji wątków za pomocą mutexów w początkowej części programu (obszar zmiennych globalnych) definiuje się uchwyt do mutexa, np.:

```
HANDLE hMutex ;
```

Następnie, w początkowej części funkcji `main` tworzy się mutex, np.:

```
hMutex = CreateMutex(NULL, FALSE, NULL);
```

Wartość drugiego parametru `FALSE` oznacza, że proces tworzący mutex nie jest jego właścicielem.

Jeśli mutex znajduje się w stanie *sygnalizowany*, to wątek za pomocą funkcji `WaitForSingleObject` może przejąć (objąć) mutex — od tej chwili mutex będzie się znajdował w stanie *niesygnalizowany*. Jeśli jednak wątek znajduje się w stanie *niesygnalizowany*, to wykonanie funkcji `WaitForSingleObject` powoduje zawieszenie

wykonywania wątku. Tak więc jeśli mutex został przejęty przez jakiś wątek, to każda próba przejęcia mutexa przez inny wątek spowoduje zawieszenie wykonywania tego wątku.

Po przejęciu mutexa wątek może wykonywać rozmaite działania dotyczące określonego zasobu, który może być używany jednocześnie tylko przez jeden wątek. Po wykonaniu działań wątek powinien zwolnić mutexa za pomocą funkcji `ReleaseMutex(hMutex)`. W trakcie wykonywania tych działań wątek ma pewność, że żaden inny wątek nie będzie mógł objąć mutexa, co oznacza że nie będzie mógł korzystać z omawianego zasobu. Przykładowa konstrukcja może być następująca:

```
WaitForSingleObject(hMutex, INFINITE);
- - - - - - - - - - - - - - - - - - - - - - -
(tu znajdują się instrukcje dotyczące
 określonego zasobu)
- - - - - - - - - - - - - - - - - - - - - - -
ReleaseMutex(hMutex) ;
```

Semafory stanowią inną technikę synchronizacji. Semafory budowane są na funkcjonalności mutexów, do których dodano zdolność zliczania. Zatem w tym samym czasie, z góry określona liczba wątków może uzyskać dostęp do synchronizowanego kodu.

Przykładowo, jeśli komputer ma trzy porty szeregowe, to najwyżej trzy wątki mogą wykonywać działania na portach szeregowych. Ta sytuacja stanowi doskonałą okazję do użycia semafora. W celu nadzorowania użycia portu szeregowego tworzony jest semafor z licznikiem równym 3. Semafor jest w stanie *sygnalizowany*, jeśli jego licznik zasobów jest większy od zera, i w stanie *niesygnalizowany*, jeśli licznik jest równy 0 (licznik nie może być nigdy mniejszy od zera). Każdorazowo, gdy wątek wykonuje funkcję `WaitForSingleObject()`, system sprawdza czy licznik dla semafora jest większy od zera. Jeśli tak, to zmniejsza licznik o 1 i wznawia wątek (proces). W przeciwnym razie system wprowadza wątek w stan uśpienia, aż do chwili gdy inny wątek zwolni zasób (czyli zwiększy licznik).

Proces, który wywołuje funkcję `CreateSemaphore()` określa początkową wartość licznika (drugi parametr) i jego wartość maksymalną (trzeci parametr). Jeśli wątek przechodzi przez semafor, to licznik jest zmniejszany o 1. Po wykonaniu działań związanych z określonym zasobem wątek zwiększa licznik semafora o 1 za pomocą funkcji `ReleaseSemaphore(hSemafor, 1, NULL)`.

Podobnie jak w przypadku mutexów, w początkowej części programu (obszar zmiennych globalnych) definiuje się uchwyt do semafora, np.:

```
HANDLE hSemafor ;
```

Następnie, w początkowej części funkcji `main` tworzy się semafor, np.:

```
hSemafor = CreateSemaphore(NULL, 1, 1, NULL);
```

Funkcja `CreateSemaphore` ma cztery parametry: atrybuty, licznik początkowy semafora, wartość maksymalna semafora, nazwa semafora (tu: niezdefiniowana).
Poniżej podano przykład wykorzystania semafora.

```
WaitForSingleObject(hSemafor, INFINITE);
- - - - - - - - - - - - - - - - - - - - - - -
(tu znajdują się instrukcje dotyczące
 określonego zasobu)
- - - - - - - - - - - - - - - - - - - - - - -
ReleaseSemaphore(hSemafor, 1, NULL) ;
```

Program przykładowy: synchronizacja za pomocą sekcji krytycznych

Pokazany niżej program przykładowy generuje i wyświetla liczby pierwsze. Z chwilą uruchomienia programu zaczyna działać wątek główny. Następnie, wątek główny za pomocą funkcji `_beginthreadex` tworzy dwa wątki (oznaczone A i B), które generują liczby pierwsze. Pierwszy wątek poszukuje liczb pierwszych wśród liczb ze zbioru 3+4n = {3, 7, 11, 15, 19, . . .}, a drugi wątek poszukuje w zbiorze 5+4n = {5, 9, 13, 17, 21, . . .}. Znalezione liczby są zapisywane w obszarze pamięci i sukcesywnie wyświetlane na ekranie przez instrukcje zawarte w wątku głównym.

Liczby pierwsze wyświetlane są porcjami po 160 liczb. Wyświetlanie liczb następuje dopiero po wpisaniu przez oba wątki kolejnej porcji 160 liczb. Fragment wątku głównego, który realizuje wyświetlanie jest blokowany przez semafor (linia: `WaitForSingleObject(wyswietl, INFINITE)`).

Ponieważ oba wątki wykonywane są równolegle, więc należy przewidywać sytuację, w której oba wątki próbowałyby jednocześnie dopisać utworzone liczby do tablicy. Wynikiem takiej operacji mogłoby być zapisanie nowej liczby na miejscu poprzedniej. W celu wyeliminowania takich sytuacji zapis liczb pierwszych do tablicy odbywa się w ramach *sekcji krytycznej*. Tylko jeden wątek może znajdować się w danej sekcji krytycznej — inne wątki będą oczekiwać na wejście, aż obecny wątek zgłosi, że ją opuszcza. Zatem zapis nowej liczby do tablicy może się rozpocząć dopiero po zapisaniu poprzedniej liczby i aktualizacji wartości wskaźnika zapisu.

Ilość generowanych liczb można podać w linii wywołania programu (jeśli wywoływany jest z poziomu okienka konsoli). Jeśli ilość nie zostanie podana, to program generuje i wyświetla 3200 liczb. Ze względu na rozdzielenie generowania liczb na dwa wątki, wartości zapisywane w obszarze wynikowym i wyświetlane na ekranie, nie są uporządkowane rosnąco.

Trzeba również dodać, że podana wersja programu generuje zadaną ilość liczb pierwszych, jednak bez gwarancji, że będą to wszystkie liczby pierwsze w określonym przedziale. Należy bowiem brać pod uwagę skrajny przypadek, gdy jeden z wątków wykonywany jest znacznie wolniej wskutek czego większość wygenerowanych liczb będzie wynikiem działania drugiego wątku (który przegląda tylko podzbiór liczb nieparzystych).. Poniżej podano kod źródłowy omawianego programu.

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


// prototyp funkcji wątku
// uwaga: przed stdcall występują dwa znaki podkreślenia
unsigned  __stdcall  liczba_pierwsza  (void  *  parametry);

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

// deklaracje zmiennych globalnych (używanych przez wszystkie
// wątki)

HANDLE  watekA,  watekB,  wyswietl;
```

```
CRITICAL_SECTION  sekcja_krytyczna;

// struktura używana do przekazywania parametrów do obu wątków
struct dane_watku
{
  unsigned  int  wartosc_poczatkowa;
  unsigned  int  wzrost;
};

unsigned int * adres_obszaru, licznik, ilosc_liczb;

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

int main(int argc, char * argv[])
{
  // to jest wątek główny uruchamiany automatycznie
  // przez system

  unsigned int i, j, ida, idb, ilosc_liczb_wyswietlonych;
  struct  dane_watku  dane_watku_A, dane_watku_B;

  printf("\nPrzyklad przetwarzania wielowatkowego\n" );

  if (argc < 2)
  {
    printf("\nW linii wywolania programu można podać żądaną
ilość liczb pierwszych\n");
     ilosc_liczb = 3200;
  }
  else
  {
    ilosc_liczb = atoi(argv[1]);
     // zwiększenie ilości wyznaczanych liczb do
     // wielokrotności 160
     if (ilosc_liczb % 160) ilosc_liczb =
                          (ilosc_liczb /160 + 1) * 160;
  }

  // wypełnianie struktur przekazywanych do wątków A i B
  dane_watku_A.wartosc_poczatkowa = 3;
  dane_watku_A.wzrost = 4;
  dane_watku_B.wartosc_poczatkowa = 5;
  dane_watku_B.wzrost = 4;

  // przydzielanie obszaru pamięci do przechowywania
  // obliczonych liczb pierwszych - ze względu na pracę
  // wielowątkową może zostać
  // wyznaczona dodatkowa liczba poza wymaganymi
  adres_obszaru = (unsigned int *) malloc((ilosc_liczb + 1)
                                  * sizeof(unsigned int));

  licznik = 0; // licznik liczb generowanych przez oba wątki
```

```
// tworzenie semafora używanego przy wyświetlaniu liczb
// drugi parametr określa początkową wartość licznika,
// a trzeci jego maksymalną wartość
wyswietl = CreateSemaphore(NULL, 0, ilosc_liczb/160 + 1,
                                                    NULL);

// inicjalizacja zmiennej dla sekcji krytycznej
InitializeCriticalSection (& sekcja_krytyczna);

// tworzenie wątków A i B
// oba wątki, w chwili uruchomienia rozpoczynają wykonywanie
// tej samej funkcji 'liczba_pierwsza', lecz z różnymi
// parametrami startowymi;
// funkcja do wykonania przekazywana jest przez
// parametr trzeci
// parametry startowe wątku przekazywane są przez
// parametr czwarty

watekA = (HANDLE) _beginthreadex (NULL, 0, &liczba_pierwsza,
                          &dane_watku_A, 0, &ida);

watekB = (HANDLE) _beginthreadex (NULL, 0, &liczba_pierwsza,
                          &dane_watku_B, 0, &idb);

// poniższy kod stanowi dalej wątek główny

ilosc_liczb_wyswietlonych = 0;

// sukcesywne wyświetlanie wygenerowanych liczb

do
{
  // oczekiwanie na semafor (gdy jest dostępna kolejna
  // grupa 160 liczb przygotowanych do wyświetlenia)
  // funkcja WaitForSingleObject zatrzymuje dalsze
  // wykonywanie wątku, jeśli wartość semafora wynosi 0
  // jeśli semafor jest > 0, to funkcja ta zmniejsza go o 1
  // i powoduje
  // kontynuowanie wykonywania programu

   WaitForSingleObject(wyswietl, INFINITE);

   // wyswietlanie 160 liczb, po 8 liczb w wierszu
   for (j = 0 ; j < 20; j++)
   {
     printf("\n");

     for (i = 0; i < 8; i++)
     {
       printf("%8d",
       * (adres_obszaru + ilosc_liczb_wyswietlonych));

       ilosc_liczb_wyswietlonych++;
     }
```

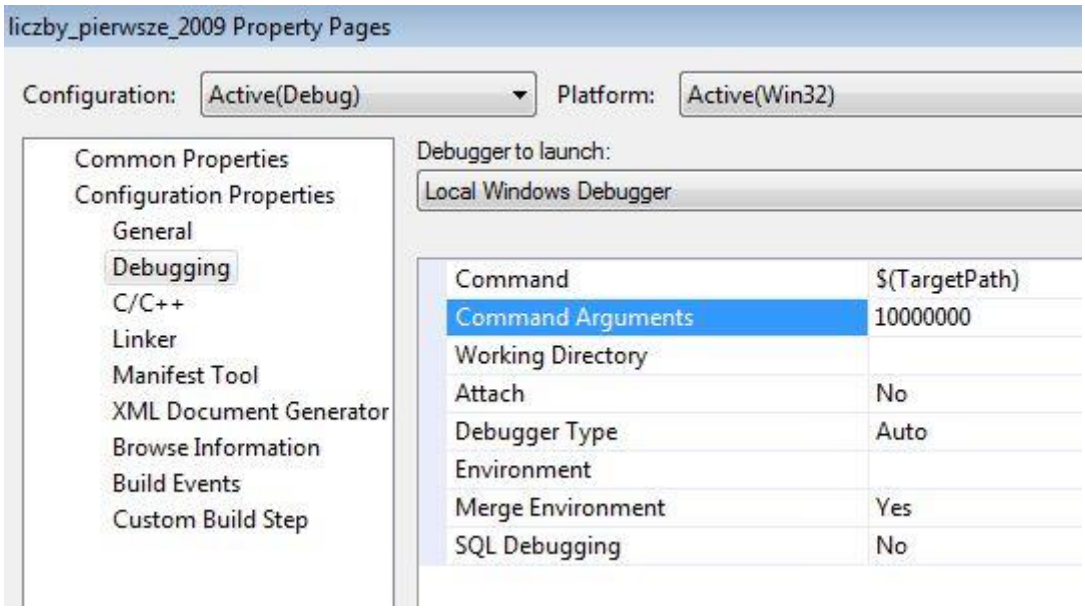
```
    }

    printf("\nLiczb wyswietlonych = %3.2f%%  %8d
(wyswietlonych)  %8d (wygenerowanych)\n",
          100.0 * ilosc_liczb_wyswietlonych/licznik,
          ilosc_liczb_wyswietlonych, licznik);

  } while (ilosc_liczb_wyswietlonych < ilosc_liczb);


  // przygotowanie do zakończenia wątku głównego (i całego
  // programu)

  CloseHandle (watekA);
  CloseHandle (watekB);

  // usunięcie sekcji krytycznej
  DeleteCriticalSection (&sekcja_krytyczna);

  return 0;
}

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

// funkcja wątku - generowanie liczb pierwszych

unsigned __stdcall liczba_pierwsza (void * parametry)
{

  unsigned int ik, ip, p1, p2;
  char a;

  // odczytanie parametrów przekazanych do wątku

  p1 = ((struct dane_watku *) parametry) ->
                              wartosc_poczatkowa;

  p2 = ((struct dane_watku *) parametry) -> wzrost;


  // generowanie liczb pierwszych poprzez wielokrotne
  // dzielenia przez kolejne liczby nieparzyste

  for (ip = p1; licznik < ilosc_liczb; ip = ip + p2)
  {
    // zmienna 'a' pełni rolę znacznika używanego do
    // sygnalizacji, że znaleziono dzielnik - oznacza
    // to, że liczba jest złożona
    a = 0;

    for (ik = 3; ik < (ip / 2); ik = ik + 2)
    {
      if ((ip % ik) == 0)
      {
```

```
        // reszta = 0, wiec liczba nie jest pierwsza
        a = 1; break;
      }
    }

    if (a==0)
    {
      // dla wszystkich dzielników reszta była różna od
      // zera, więc liczba jest pierwsza

      // oczekiwanie na wejście do sekcji krytycznej
      EnterCriticalSection (& sekcja_krytyczna);

      // zapisanie liczby pierwszej w obszarze pamięci
      adres_obszaru[licznik] = ip;
      licznik++; // inkrementacja licznika liczb pierwszych

      // inkrementacja semafora, jeśli dostępny jest kolejny
      // blok 160 liczb pierwszych
       if ((licznik % 160) == 0)
                        ReleaseSemaphore(wyswietl, 1, NULL);

       // wyjście z sekcji krytycznej
       LeaveCriticalSection (& sekcja_krytyczna);
     }
  }
  _endthreadex(0);
  return 0;
}
```

Uruchamianie programów z parametrami podanymi w linii wywołania

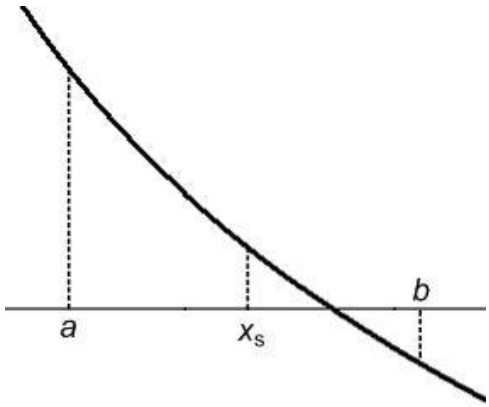
Opisany powyżej program generujący liczby pierwsze opcjonalnie można uruchomić z parametrem podanym w linii wywołania. Jeśli parametr został pominięty, to program przyjmuje domyślną wartość parametru równą 3200. Jeśli program wywoływany jest z okienka konsoli, to parametr podaje się bezpośrednio za nazwą uruchamianego programu, np. `watki 640`.

W przypadku uruchamiania programu z poziomu MS Visual Studio należy kliknąć prawym klawiszem myszki w oknie Solution Explorer na nazwę projektu, wybrać opcję Properties, następnie Debugging i do wiersza Command Arguments wpisać odpowiednią wartość, tak jak pokazano na poniższym rysunku.



Przypomnienie algorytmu bisekcji

Algorytm bisekcji pozwala na odnalezienie, w pewnym przedziale wartości, takiego argumentu funkcji *f*, dla którego funkcja przyjmuje wartość bliską zera lub równą zero. Punktem wyjścia algorytmu jest znajomość dwóch wartości *a* i *b* takich, że w przedziale (*a*, *b*) funkcja *f* jest ciągła i zachodzi związek $f(a) * f(b) < 0$ (wartości funkcji na końcach rozpatrywanego przedziału mają różne znaki). Obliczenia kończymy jeśli wartość bezwzględna funkcji dla wyznaczonego argumentu jest mniejsza lub równa od pewnej zadanej dokładności ε (np. = 0.0001). Realizacja algorytmu obejmuje następujące czynności:



1. Wyznaczamy środek aktualnego przedziału wg formuły:

$$x_s = 0.5 * (a + b)$$

2. Obliczamy wartość funkcji w środku przedziału $y = f(x_s)$
3. Jeśli wartość $|y| \leq \varepsilon$, to kończymy obliczenia przyjmując, że x_s jest poszukiwaną wartością, w przeciwnym razie przechodzimy do punktu 4.
4. Obliczamy teraz wartość wyrażenia $f(a) * f(x_s)$ — jeśli wyrażenie to jest ujemne, to znaczy że na końcach przedziału (a, x_s) funkcja przyjmuje różne znaki, więc trzeba podstawić $b = x_s$, natomiast jeśli iloczyn $f(a) * f(x_s)$ jest dodatni, to oznacza, że miejsce zerowe znajduje się w przedziale (x_s, b), więc musimy podstawić $a = x_s$
5. Powtarzamy opisane operacje począwszy od punktu 1.

Zadania do wykonania

1. Uruchomić podane wcześniej trzy programy przykładowe (str. 5, str. 8-10, str. 12-16). Programy uruchomić w środowisku MS Visual Studio 2010 jako aplikacje konsolowe tworząc dla każdego programu oddzielny projekt.

   Uwaga: zdarza się, że podczas kompilacji programu przykładowego pojawiają się ostrzeżenia (ang. warnings) dotyczące niekompatybilności typu LPCWSTR, a następnie po uruchomieniu aplikacji w oknie `MessageBox` pojawiają się nieczytelne znaki. Przyczyną jest zazwyczaj ustawienie opcji konfiguracyjnej Use Unicode Character Set. Opcja ta jest dostępna poprzez kliknięcie (w oknie Solution Explorer) prawym klawiszem myszki na nazwę projektu, następnie wybranie Properties i dalej Configuration Properties / General / Character Set. Opcja ta powinna być ustawiona w postaci „Not Set".

2. Uruchomić program wyznaczający wartości liczb pierwszych z parametrem o dużej wartości (np. 10 milionów). Włączyć *Menedżer zadań Windows*, wybrać zakładkę *Wydajność* i zaobserwować obciążenie procesorów w trakcie wykonywania programu.

3. Wzorując się na programie przykładowym wyznaczania liczb pierwszych, napisać program wielowątkowy w języku C, który wyznaczy miejsca zerowe funkcji podanej przez prowadzącego zajęcia. Przyjmujemy, że funkcja ma kilka miejsc zerowych, a ich wyznaczanie powinno być realizowane równolegle przez kilka wątków, z których każdy zajmuje poszukiwaniem w innym przedziale. Do wyznaczenia miejsc zerowych zastosować metodę *bisekcji* (zob opis na poprzedniej stronie).

   Zalecenia dodatkowe do zadania 3.

   1. Do funkcji wątku należy przekazywać dwa parametry *a* i *b* określające granice przedziału, w którym należy poszukiwać miejsca zerowego. Podane parametry przekazuje się w postaci struktury — wskaźnik do tej struktury jest argumentem funkcji wątku (dalsze szczegóły podane są obok przykładu obliczania pola prostokąta).
   2. W podanym przedziale funkcja musi być ciągła i musi zachodzić związek f(*a*) * f(*b*) < 0.
   3. W pierwszej, uproszczonej wersji programu funkcja wątku powinna jedynie wyświetlać wartości parametrów *a* i *b*. Po upewnieniu się, że parametry te przekazywane są poprawnie można przystąpić do kodowania algorytmu bisekcji.
   4. Wygodnie jest określić wymaganą dokładność obliczeń za pomocą stałej, np.:

      ```
      #define    DOKLADNOSC      0.000001
      ```

   5. Użycie wielu typowych funkcji trygonometrycznych, logarytmicznych itp. wymaga wprowadzenia do programu pliku nagłówkowego `<math.h>`. Zazwyczaj deklarację tego pliku poprzedza jeszcze deklaracja stałej:

      ```
      #define    _USE_MATH_DEFINES
      #include   <math.h>
      ```

      Po wprowadzeniu powyższych wierszy można m.in. używać symbolu `M_PI`, który reprezentuje liczbę π (3.14...) w postaci zaokrąglonej do 21 cyfr, np. `P = M_PI * r * r` . Podobnie symbol `M_E` reprezentuje liczbę e (2.71...), a symbol `M_SQRT2` oznacza pierwiastek z 2 (1.41...).
   6. Po wyznaczeniu miejsca zerowego wątek powinien wyświetlić znalezioną wartość na ekranie. Używana do tego celu funkcja `printf()` musi być traktowana jako

pewnego rodzaju zasób, do którego dostęp może mieć tylko jedne wątek — zapobiega to możliwości jednoczesnego wyświetlania wyników przez kilka wątków. Dostęp do zasobu można ograniczyć za pomocą mutexa, semafora lub sekcji krytycznej.

7. Wątki nie powinny dublować swojej pracy, każdy szuka *innych* miejsc zerowych.
8. Obliczenia należy wykonywać z użyciem typu `float` lub `double`. Do wyznaczania wartości bezwzględnej zmiennych typu `double` używa się funkcji `fabs`, albo funkcji `fabsf` w przypadku typu `float`.
9. Przykładowo, zadanie podane przez prowadzącego może polegać na wyznaczeniu miejsc zerowych funkcji $y = x^3 + 4x^2 - 25x - 28$. Można przyjąć, że pierwszy wątek będzie poszukiwał miejsca zerowego w przedziale <–10, –5>, drugi wątek w przedziale <–2, +2>, a trzeci wątek w przedziale <+2, +7>. Wykres omawianej funkcji pokazany jest na rysunku.