Podstawy programowania współbieżnego

Opracowanie: Tomasz Krzeszowski
Politechnika Rzeszowska,
Katedra Informatyki i Automatyki,
Rzeszów, 2010

1. Wprowadzenie

Programowanie współbieżne to technika pisania programów w których wiele czynności może być wykonywanych jednocześnie. W programach tego typu dokonuje się podziału zadań na mniejsze spójne fragmenty kodu, które zwane są wątkami. Wątki są więc zbiorem operacji, które mogą być wykonane niezależnie od innych operacji wykonywanych podczas danego zadania. Przykładem programów wykorzystujących współbieżność są różnego rodzaju aplikacje okienkowe oraz systemy operacyjne, w których współbieżność pozwala na uzyskanie wrażenie, że wszystkie zadania są realizowane równocześnie. Na komputerach jednoprocesorowych (z procesorami jednordzeniowymi) wątki są uruchamiane na przemian, natomiast w komputerach wieloprocesorowych (lub w przypadku procesorów wielordzeniowych) wiele wątków może być wykonywanych jednocześnie. Zastosowanie technik programowanie współbieżnego pozwala uzyskanie znacznego przyspieszenia obliczeń na komputerach wieloprocesorowych oraz komputerach wyposażonych w procesory wielordzeniowe. Oprócz wspomnianych zalet należy także nadmienić, że pisanie programów współbieżnych jest znacznie trudniejsze niż przygotowywanie programów sekwencyjnych (programy w których operacje są wykonywane jedna po drugiej). Do najczęściej występujących problemów należą problemy związane z synchronizacją wykonywania poszczególnych wątków oraz współdzielenie zasobów przez wątki.

2. Podstawowe pojęcia

Proces – jest reprezentacją programu w systemie operacyjnym, każdy utworzony proces otrzymuje unikalny identyfikator (PID), oraz pewne zasoby (m.in. pamięć i czas procesora). W ramach danego procesu może zostać utworzonych wiele wątków, które wykonują określone części programu. Za

współbieżne działanie procesów oraz wątków odpowiada system operacyjny. Każdy proces ma do dyspozycji przydzielone niezależne zasoby.

Wątek (ang. *thread*) – jest to część programu wykonywana współbieżnie. W jednym procesie (programie) może istnieć wiele wątków. Wszystkie wątki pracujące w obrębie danego procesu współdzielą jego zasoby, tzn. mają dostęp do tej samej przestrzeni adresowej oraz innych zasobów. Umożliwia to łatwą komunikację pomiędzy poszczególnymi wątkami danego procesu.

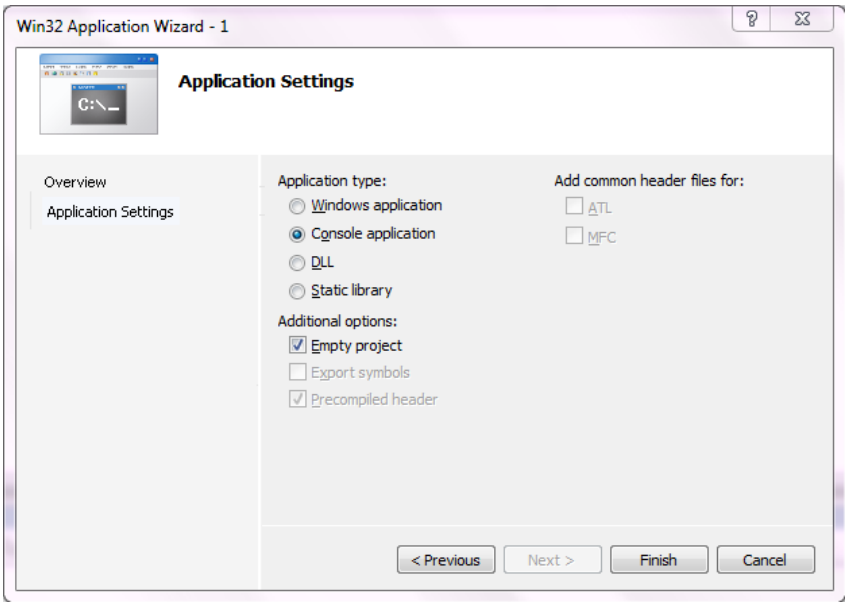
Priorytet – każdy wątek oraz proces ma określony priorytet. Priorytet określa jak często dany wątek (proces) powinien otrzymywać czas procesora. Podczas działania programu priorytet może ulec zmianie.

3. Konfiguracja środowiska Visual Studio

Obecnie język C/C++ nie wspiera obsługi wątków, dlatego w przedstawionych przykładach wykorzystano bibliotekę **Boost**, która umożliwia pisanie programów wielowątkowych. W celu uruchomienia przedstawionych w dalszej części przykładów należy pobrać plik **boost_thread.zip**, a następnie odpowiednio skonfigurować kompilator Visual Studio. Zawarte w pliku **boost_thread.zip** biblioteki można wykorzystać w środowisku Visual Studio w wersjach 7.1, 8.0 oraz 9.0.

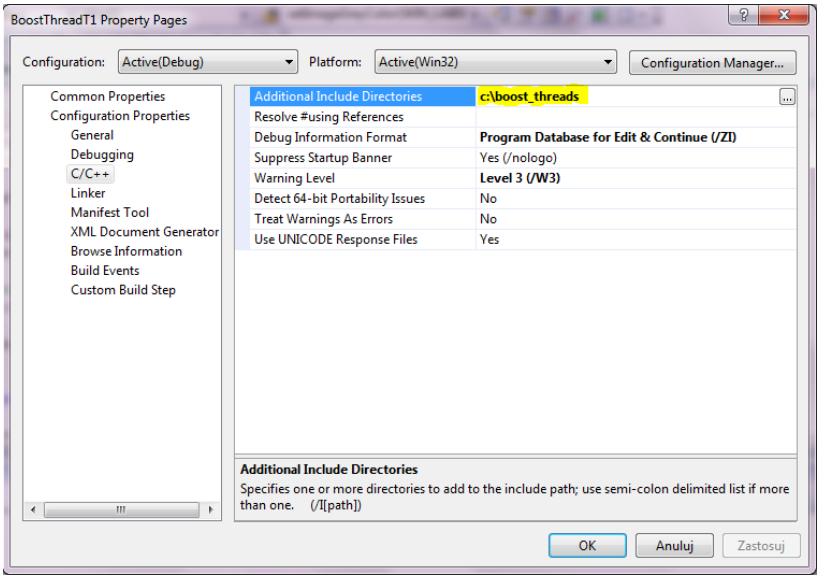
W celu kompilacji aplikacji wykorzystujących wątki należy wykonać kolejno (w poniższej instrukcji konfiguracji dokonano na przykładzie środowiska Visual Studio 8.0, w Visual Studio 7.1 oraz 9.0 konfiguracja przebiega w analogiczny sposób):

- wypakować plik z bibliotekami **Boost-a** na dysk;
- uruchomić wybrane środowisko Visual Studio
- utworzyć pustą aplikację konsolową (w opcjach projektu należy zaznaczy: „Console application” oraz „Empty project", zob. Rys. 1)



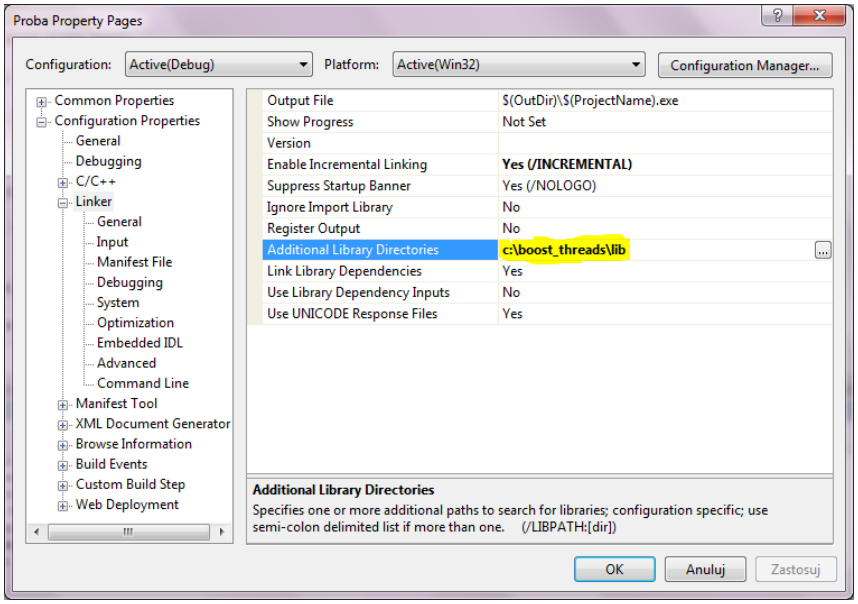
Rys. 1 Prawidłowe ustawienia projektu

- dodać nowy plik .cpp do projektu (np. main.cpp)
- włączyć okno z opcjami projektu (kliknąć prawym przyciskiem myszy na nazwie projektu w oknie „Solution Explorer" i wybrać z menu „Properties"
- w zakładce „Configuration Properties > C/C++ > General > Additional Include Directories" należy wprowadzić ścieżkę do katalogu głównego biblioteki Boost np. jeśli plik boost_thread.zip został wypakowany do katalogu o tej samej nazwie na dysku C to prawidłowa ścieżka będzie miała postać: c:\boost_threads (rys. 2)



Rys. 2 Ustawienie ścieżki do katalogu z biblioteką

- następnie należy wpisać w „Configuration Properties > Linker > Additional Library Directories” ścieżkę do katalogu z plikami .lib biblioteki np. c:\boost_threads\lib (rys. 3)



Rys. 3 Ustawienie ścieżki do plików .lib

- na koniec należy sprawdzić czy „Configuration Properties > C/C++ > Precompiled Headers” jest ustawiona na „Not Using Precompiled Header”, jeśli nie to zmienić na odpowiednią wartość
- po wykonaniu powyższych czynności należy skopiować przedstawiony na listingu 1 szablon programu i skompilować

Listing 1 Szablon programu

```
// dolaczenie naglowka biblioteki boost
#include <boost/thread.hpp>

int main(int argc, char* argv[])
{
    // kod uzytkownika

    // komunikat dla uzytkownika aby wcisna „enter”
    // aby zakonczyc dzialanie aplikacji
    std::cout << "Wcisnij enter aby zakonczyc" << std::endl;
    getchar();

    return 0;
}
```

Jeśli kompilacją przebiegnie bez żadnych błędów można przejść do dalszej części ćwiczenia.

4. Tworzenie wątków

Wątki w bibliotece **Boost** są reprezentowane przez klasę `boost::thread`. Przeciążony konstruktor klasy wątku pobiera jako parametr adres funkcji, której kod ma być wykonywany w wątku. Przykład utworzenia wątku:

```
boost::thread nazwa_watku(&nazwa_funkcji);
```

Funkcja `main()` jest wykonywana w głównym wątku programu. Po utworzeniu nowego wątku przy pomocy klasy `boost::thread` w programie działają już niezależnie dwa wątki. Jeśli chcemy aby jeden wątek poczekał na zakończenie innego wątku to korzystamy z funkcji `boost::thread::join()`. Wątek w którym została wywołana funkcja `join()` oczekuje na zakończenie wątku dla którego wywołano omawianą funkcję. Na listingu 2 przedstawiono przykładowy program, w którym tworzony jest wątek (`watek`) wypisujący komunikat na ekran. W wątku głównym programu wywoływana jest funkcja `join()` dla utworzonego wcześniej obiektu wątku (`watek`), zapewnia ona poprawną kolejność wypisania komunikatów dla użytkownika.

Listing 2 Prosty program wykorzystujący wątki

```
// dolaczenie naglowka biblioteki boost
#include <boost/thread.hpp>
// funkcja ktorej kod bedzie wykonywany w watku
void witaj()
{
    // wypisanie komunikatu
    std::cout << "Witaj" << std::endl;
}
```

```
int main(int argc, char* argv[])
{
    // utworzenie nowego watku i uruchomienie
    boost::thread watek(&witaj);
    // oczekiwanie na zakonczenie watku
    watek.join();

    std::cout << "Wcisnij enter aby zakonczyc" << std::endl;
    getchar();

    return 0;
}
```

Do funkcji wątków można także przekazywać parametry, jest to realizowane z wykorzystaniem funkcji `boost::thread::bind(f, a1, a2, …)`, która przyjmuję jako parametry adres funkcji oraz listę parametrów.

Listing 3 Przykład wykorzystania funkcji wątków z parametrami

```
#include <iostream>
#include <boost/thread.hpp>
#include <boost/date_time.hpp>

// funkcja watku, jako parametry przyjmuje nazwe watku oraz
// modyfikator czasu na jaki watek będzie usypiany
void funkcjaWatku(std::string nazwaWatku, float a)
{
    for(int i=1; i<1000; ++i)
    {
        // wylosowanie liczby z przedzialu od 0 do 5 razy modifikator
        float czas = ( (rand()%100)/50.0f)*a;
        // ustawienie czasu uspienia
        boost::posix_time::seconds czasUspienia(czas);

        // w co dziesiatej iteracji nastepuje uspienie
        if(!(i%10))
        {
            std::cout << "Uspienie watku: " << nazwaWatku
                << " na czas " << czas << "s" << std::endl;
            // uspienie watku
            boost::this_thread::sleep(czasUspienia);
            //std::cout << "Watek wybudzil sie." << std::endl;
        }

        // wypisanie komunikatu
        std::cout << "Jestem watek o nazwie: " << nazwaWatku << std::endl;
    }
}

int main(int argc, char* argv[])
{
    // inicjalizacja generatora liczb losowych
```

```
	srand(time(NULL));

	// utworzenie watkow
	boost::thread watek1(boost::bind(&funkcjaWatku,"Watek 1", 1.0f));
	// watek bedzie usypiany na 3 razy dluzszy czas niz normalnie
	boost::thread watek2(boost::bind(&funkcjaWatku,"Watek 2", 3.0f));
	boost::thread watek3(boost::bind(&funkcjaWatku,"Watek 3", 1.0f));
	// watek bedzie usypiany na 0.5 razy dluzszy czas niz normalnie
	boost::thread watek4(boost::bind(&funkcjaWatku,"Watek 4", 0.5f));

	// oczekiwanie na zakoczenie
	watek1.join();
	watek2.join();
	watek3.join();
	watek4.join();

	std::cout << "Wcisnij enter" << std::endl;
	getchar();
	return 0;

}
```

5. Zarządzanie wątkami

Przedstawione zostaną teraz funkcję służące do zarządzania wątkami. W pierwszej kolejności zostanie zaprezentowana funkcja `sleep`, która umożliwia zatrzymanie działania wątku na pewien określony czas.

`void boost::thread::sleep(system_time const& abs_time).` Funkcja ta przyjmuje jako parametr wartość czasu na jaki ma zostać uśpiony wątek. Przykład wykorzystania przedstawia listing 3.

Listing 4 Użycie funkcji `sleep`

```
#include <iostream>
#include <boost/thread.hpp>
#include <boost/date_time.hpp>

void obliczenia()
{
	for(int i=1; i<1000; ++i)
	{
		// obliczenia ktore maja byc wykonywane w watku
		float a = 0;
		for(int j=0; j<1000000; ++j) { a = (a+sqrt(i*3.5f)*500.0f)/50.0f;}
		// wypisanie wyniku
		std::cout << "Watek 1 " << a << std::endl;
	}
	std::cout << "Watek z obliczeniami zakonczyl sie" << std::endl;
}
```

```
// funkcja w ktorej bedzie nastepowalo uspienie watku
void uspienie()
{
     // ustawienie czasu uspienia na 2s
     boost::posix_time::seconds czasUspienia(2);

     for(int i=0; i<600; ++i)
     {
          float a=0.0f;
          //jakies obliczenia
          for(int j=0; j<1000000; ++j)
          {
               a=sqrt(pow(50.0f,4.0f)*5.0f);
          }
          std::cout << "Watek 2 " << a << std::endl;

          // w co dziesiatej iteracji nastepuje uspienie
          if(!(i%10))
          {
               std::cout << "Uspienie watku." << std::endl;
               // uspienie watku
               boost::this_thread::sleep(czasUspienia);
               std::cout << "Watek wybudzil sie." << std::endl;
          }


     }
    std::cout << "Watek ktory byl usypiany zakonczyl sie." << std::endl;
}

int main(int argc, char* argv[])
{
     // utworzenie watkow
     boost::thread watek1(&obliczenia);
     boost::thread watek2(&uspienie);

     // oczekiwanie na zakoczenie
     watek1.join();
     watek2.join();

     std::cout << "Wcisnij enter" << std::endl;
     getchar();
     return 0;
}
```

Zostanie teraz omówiona funkcja `yield()`, która pozwala na oddanie czasu procesora innemu wątkowi. Po wywołaniu funkcji `yield()`, dany wątek przestaje być wykonywany, a jego czas procesora zostaje przekazany innemu wątkowi. Należy uruchomić przykład z listingu 5, zaobserwować działanie programu, a następnie uruchomić ten sam przykład z odkomentowaną linią, w której jest wywołana funkcja `yield()`. Po odkomentowaniu wywołania omawianej funkcji, wątek będzie znacznie rzadziej wypisywał komunikat

ponieważ po wykonaniu każdego obliczenia będzie oddawał swój czas procesora innym wątkom.

Listing 5 Użycie funkcji `yield()`

```
#include <iostream>
#include <boost/thread.hpp>

void obliczenia()
{
	for(int i=1; i<1000; ++i)
	{
		// obliczenia ktore maja byc wykonywane w watku
		float a = 0;
		for(int j=0; j<1000000; ++j)
		{
			a = (a+sqrt(i*3.5f)*500.0f)/50.0f;
		}
		// wypisanie komunikatu
		std::cout << "Watek 1 " << a << std::endl;
	}
	std::cout << "Watek z obliczeniami zakonczyl sie" << std::endl;
}

// funkcja w ktorej bedzie wywolywana funkcja yield
void obliczeniaZyield()
{
	for(int i=0; i<1000; ++i)
	{
		float a=0.0f;
		//jakies obliczenia
		for(int j=0; j<1000000; ++j)
		{
			a = (a+sqrt(i*3.5f)*500.0f)/50.0f;
			//oddanie czasu procesora (odkomentowac)
			//boost::thread::yield();
		}
		//wypisanie komunikatu
		std::cout << "Watek 2 " << a << std::endl;
	}
}

int main(int argc, char* argv[])
{
	// utworzenie watkow
	boost::thread watek1(&obliczenia);
	boost::thread watek2(&obliczeniaZyield);

	// oczekiwanie na zakoczenie
	watek1.join();
	watek2.join();

	std::cout << "Wcisnij enter" << std::endl;
	getchar();
	return 0;
}
```

6. Synchronizacja wątków

Bardzo ważnym aspektem programowania aplikacji wielowątkowych jest współdzielenie zasobów. W celu zapewnienia poprawnego działania aplikacji należy zadbać o synchronizację dostępu do zasobów współdzielonych przez wątki. W przypadku braku synchronizacji dostępu może się zdarzyć sytuacja, w której kilka wątków będzie jednocześnie zapisywać dane do zmiennej współdzielonej w wyniku czego otrzymamy błędne wyniki. W bibliotece **Boost** synchronizacja jest realizowana poprzez zastosowanie tzw. mutex-ów. Są to obiekty, które pozwalają na zablokowanie dostępu do wybranego fragmentu kodu. Podczas synchronizacji możemy wyróżnić dwa główne etapy: zamknięcie obiektu mutex oraz otwarcie obiektu mutex. Należy pamiętać, aby każda nałożona blokada została zdjęta w innym przypadku może dojść do tzw. „zakleszczenia". Jest to sytuacja, w której wątki wzajemnie się blokują co powoduje wstrzymanie działania programu. Na listingu 6 przedstawiono prosty przykład synchronizacji dostępu do wspólnej zmiennej z wykorzystaniem obiektu `boost::mutex`. W przykładzie tym do synchronizacji wykorzystano obiekt `boost::mutex::scoped_lock`. W wypadku zastosowania tego obiektu blokada jest automatycznie podnoszona w konstruktorze oraz opuszczana w destruktorze, tzn. że blokada zostanie zdjęta automatycznie po opuszczeniu zasiegu, w którym utworzono mutex. Dostęp do urządzeń wejścia/wyjścia także powinien być synchronizowany. Synchronizacja operacji wejścia/wyjścia została pominięta we wczesniejszych przykładach w celu ułatwienia analizy. W dalszej części instrukcji dostęp do urządzeń wejścia/wyjścia będzie synchronizowany za pomocą obiektu mutex.

Listing 6 Synchronizacja wątków – prosty przykład

```
#include <iostream>
#include <boost/thread.hpp>
#include <boost/thread/mutex.hpp>

// obiekt synchronizujacy
```

```
boost::mutex mutex;
// wspoldzielona zmienna
float zmiennaWspoldzielona = 0;

void obliczenia(int id)
{
     for(int i=1; i<1000; ++i)
     {
          // obliczenia ktore maja byc
          //wykonywane w watku na wspolnych zasobach
          float a = 0;
          for(int j=0; j<1000000; ++j)
          {
               a = (a+sqrt(i*3.5f)*500.0f)/50.0f;
          }
          //zablokowanie dostepu do wspolnych zasobow
          boost::mutex::scoped_lock blokada(mutex);
          zmiennaWspoldzielona += 1;
          // wypisanie komunikatu
          std::cout << "Watek " << id << ": "
               << zmiennaWspoldzielona << " a: " << a << std::endl;
     }
     std::cout << "Watek z obliczeniami zakonczyl sie" << std::endl;
}

int main(int argc, char* argv[])
{
     zmiennaWspoldzielona = 0;
     boost::thread watek1(boost::bind(&obliczenia, 1));
     boost::thread watek2(boost::bind(&obliczenia, 2));

     watek1.join();
     watek2.join();

     std::cout << "Wcisnij enter" << std::endl;
     getchar();

     return 0;
}
```

Przedstawiony zostanie teraz problem producenta i konsumenta, będący klasycznym problemem synchronizacji zasobów. W zagadnieniu tym występują dwa rodzaje procesów (konsumenci i producenci), pomiędzy którymi są współdzielone pewne zasoby (pudełko). Zadaniem producenta jest dostarczanie zasobów do pudełka, natomiast zadaniem konsumenta jest pobieranie zasobów z pudełka. W przykładzie tym wykorzystano funkcje obiektu `boost::mutex`, które umożliwiają ręczne zamknięcie (funkcja `lock()`) oraz otwarcie blokady (funkcja `unlock()`). W przypadku użycia tych funkcji należy uważać, aby ilość wywołań funkcji otwierającej blokadę, była taka sama jak ilość wywołań funkcji zamykającej blokadę. W przeciwnym wypadku może dojść

„zakleszczenia” lub wyrzucenia wyjątku. Przykład z listingu 7 przedstawia także wykorzystanie obiektu typu `boost::condition`, który udostępnia podstawowe funkcje do komunikacji pomiędzy wątkami. Funkcja `wait()` obiektu `boost::condition` pobiera jako parametr obiekt typu `boost::mutex` i zatrzymuje działanie wątku do czasu, aż inny wątek prześle odpowiednie powiadomienie. Po wywołaniu funkcji `wait()`, blokada obiektu podanego jako parametr jest ściągana. Po otrzymaniu powiadomienia od innego wątku następuje wznowienie działania wątku oraz ponowne ustawienie blokady na obiekcie `boost::mutex`. Do komunikacji pomiędzy wątkami służy metoda `notify_one()` obiektu `boost::condition`. Wznawiane są wszystkie wątki, które wywołały wcześniej metodę `wait()` danego obiektu `boost::condition`.

Listing 7 Problem producenta i konsumenta

```
#include <boost/thread/thread.hpp>
#include <boost/thread/mutex.hpp>
#include <boost/thread/condition.hpp>
#include <iostream>

// ilosc elementow w pudelku
#define ILOSC_ELEMENTOW 10
// ilosc iteracji petli
#define ILOSC_ITERACJI 100

// blokada dostepu do urzadzen wejscia/wyjscia
boost::mutex ioMutex;

class Pudelko
{
private:
     // obiekt blokujacy dostep do "zawartosci pudelka"
     boost::mutex mutex;
     // obiekt do kontoroli zaleznosci pomiedzy watkami
     boost::condition cond;

     // p - indeks okreslajacy do ktorego elementu
     // tablicy "zawartosc" wkladamy nowy element
     // c - indeks okreslajacy z ktorego elementu
     // tablicy "zawartosc" wyciagamy nowy element
     // iloscElem - aktualna ilosc elementow
     // w tablicy zawierajacej elementy pudelka
     unsigned int p, c, iloscElem;
     // zawartosc pudelka
     int zawartosc[ILOSC_ELEMENTOW];
```

```
public:
     // konstruktor bezargumentowy
     Pudelko()
          : p(0), c(0), iloscElem(0)
     {
     }
    // funkcje umozliwiajaca wlozenie do "pudelka"
     void wloz(int m)
     {
          // zablokowanie pudelka
          mutex.lock();
          //jesli pudelko jest pelne to musimy poczekac
          if (iloscElem == ILOSC_ELEMENTOW)
          {
               {
                    // blokada urzadzen we/wy
                    boost::mutex::scoped_lock lock(ioMutex);
                    std::cout << "Pudelko jest pelne. Czakamy..."
                         << std::endl;
               }
               // czekamy na oproznienie pudelka
               while (iloscElem == ILOSC_ELEMENTOW)
               {
                    cond.wait(mutex);
               }
          }
          // wstawienie nowego elementu do srodka pudelka
          this->zawartosc[p] = m;
          // wyliczenie nowego indeksu do wlozenia kolejnego elementu
          p = (p+1) % ILOSC_ELEMENTOW;
          // zwiekszenie liczby elementow w pudelku
          ++iloscElem;
          // powiadomienie pozostalych watkow o wyciagnieciu
          // elementu z pudelka
          cond.notify_one();
          //sciagniecie blokady
          mutex.unlock();
     }
     // funkcja umozliwiajaca wyciagniecie z "pudelka"
     int wyciag()
     {
          // zablokowanie pudelka
          mutex.lock();
          //jesli pudelko jest puste to musimy poczekac
          if (iloscElem == 0)
          {
               {
                    // blokada urzadzen we/wy
                    boost::mutex::scoped_lock lock(ioMutex);
                    std::cout << "Pudelko jest puste. Czekamy..."
                         << std::endl;
               }
               // czekamy na wlozenie elementow do pudelka
               while (iloscElem == 0)
               {
                    cond.wait(mutex);
               }
          }
          // wyciagniecie elementu z pudelka
          int i = this->zawartosc[c];
          // wyliczenie nowego indeksu do wyciagniecia kolejnego
```

```
            // elementu z pudelka
            c = (c+1) % ILOSC_ELEMENTOW;
            // zmniejszenie liczby elementow w pudelku
            --iloscElem;
            // powiadomienie pozostalych watkow o wyciagnieciu
            // elementu z pudelka
            cond.notify_one();
            // zdjecie blokady
            mutex.unlock();
            // zwrocenie elementu wyciagnietego z pudelka
            return i;
      }
};

// utworzenie nowego obiektu "pudelka"
Pudelko pudelko;

// funkcja watku uzupelniajacego zawartosc pudelka (producent)
void wkladanieDoPudelka()
{
      for (int n = 0; n < ILOSC_ITERACJI; ++n)
      {
            {
                  // zablokowanie dostepu do urzadzen we/wy
                  boost::mutex::scoped_lock lock(ioMutex);
                  std::cout << "wysylanie: " << n << std::endl;
            }
            // wlozenie elementu do pudelka
            pudelko.wloz(n);
      }
}
// funkcja watku pobierajacego zawartosc pudelka (konsument)
void wyciaganieZPudelka()
{
      for (int x = 0; x < ILOSC_ITERACJI; ++x)
      {
            // wyciagniecie elementu z pudelka
            int n = pudelko.wyciag();
            {
                  // zablokowanie dostepu do urzadzen we/wy
                  boost::mutex::scoped_lock lock(ioMutex);
                  std::cout << "odbieranie: " << n << std::endl;
            }
      }
}

int main(int argc, char* argv[])
{
      // utworzenie watku konsumenta i producenta
      boost::thread konsument(&wyciaganieZPudelka);
      boost::thread producent(&wkladanieDoPudelka);

      konsument.join();
      producent.join();

      std::cout << "Wcisnij enter" << std::endl;
      getchar();
      return 0;
}
```

7. Zadania do samodzielnej realizacji

7.1. Należy napisać program, w którym działają jednocześnie cztery wątki. W każdym wątku w pętli wypisywana jest informacja o identyfikatorze wątku (identyfikator przekazać do wątku jako parametr) oraz wynik obliczeń wykonywanych w pętli.

7.2. Napisać program, w którym uruchamiane są trzy wątki, a w każdym wątku wykonywane są obliczenia. Trzeci z wątków może być uruchomiony dopiero po zakończeniu działania dwóch pierwszych.

7.3. Przygotować program, w którym dwa wątki wykonują w pętli obliczenia na liczbach całkowitych. Każdy z wątków jest usypiany na czas dwóch sekund jeśli wynik obliczeń jest liczba parzystą.

7.4. Napisać program, w którym dwa wątki współdzielą zasoby. Pierwszy wątek wykonuje w pętli na współdzielonej zmiennej obliczenia, natomiast drugi wątek w pętli wypisuje aktualną wartość współdzielonej zmiennej. Należy zadbać o odpowiednią synchronizację dostępu do zasobów.

8. Literatura

[1] http://www.boost.org/doc/libs/1_44_0/doc/html/thread.html

[2] http://aszt.inf.elte.hu/~gsd/klagenfurt/material/ch03s06.html

[3]http://antonym.org/2009/05/threading-with-boost---part-i-creating-threads.html

[4]http://www.drdobbs.com/cpp/184401518;jsessionid=3RKCUFS0GP25VQE1GHRSKHWATMY32JVN?pgno=4

[5] http://www.paulbridger.com/home/