Współdzielenie obiektów w aplikacjach współbieżnych

Używanie tych samych obiektów przez różne wątki może prowadzić do niezdefiniowanego zachowania się aplikacji. Obiekty synchronizacyjne pozwalają eliminować te zjawiska. Ich użycie pokazujemy na przykładzie singletona.

Dowiesz się:

- Jak tworzyć przenośne aplikacje współbieżne w C++;
- Jak posługiwać się obiektami synchronizującymi;
- Co to jest wzorzec singletona;
- Co to jest wzorzec podwójnego sprawdzania.

Powinieneś wiedzieć:

- Jak pisać proste programy w C++;
- Co to są wątki.

Program współbieżny jest zbiorem sekwencyjnych ciągów instrukcji, które są wykonywane równolegle. Wątek reprezentuje niezależny ciąg instrukcji w ramach procesu (programu). Wątki są dostarczane przez większość systemów operacyjnych. Różne wątki tej samej aplikacji współdzielą przestrzeń adresową, więc wspólne dla wątków są: obiekty globalne, obiekty dynamiczne, obiekty automatyczne, jeżeli dostęp do nich jest realizowany przez wskaźnik lub referencję, zasoby systemu operacyjnego (pliki, okna itp.) oraz kod wykonywalny. Unikalne dla wątku są zmienne automatyczne, ponieważ dostęp do tych zmiennych wykorzystuje stos, a ten jest różny, dla różnych wątków. Program ma zawsze jeden wątek, nazywany wątkiem głównym lub inicjującym, jest to wątek wykonujący funkcję `main`. Możemy tworzyć dodatkowe wątki, nazywane wątkami obliczeniowymi. W tym artykule do reprezentowania wątków używamy klas dostarczanych przez bibliotekę boost::thread (patrz ramka).

Różne wątki mogą czytać i pisać te same obiekty dynamiczne albo globalne i w ten sposób wymieniać informacje. Ponieważ brak jest deterministycznej zasady pozwalającej określić, która z dwóch instrukcji wykonywanych w różnych wątkach wykona się szybciej, przy pisaniu i czytaniu tego samego obiektu można uzyskać różne wyniki dla różnej, rzeczywistej kolejności wykonywania instrukcji. Takie zjawisko nazywa się wyścigiem, prowadzi ono do niezdefiniowanego zachowania aplikacji. Przykładowo, jeżeli jeden wątek chce zapisać wartość A do pewnego obiektu, natomiast drugi wątek chce zapisać wartość B, w zależności od tego, w jakiej kolejności będą one pisały, w obiekcie będzie przechowywana wartość B (jeżeli wątek pierwszy wykonał operację zapisu przed wątkiem drugim), albo wartość A (jeżeli było odwrotnie, to znaczy wątek pierwszy zapisał wartość po wątku drugim), patrz Rysunek 1.

Wyścigi są specyficzne dla aplikacji współbieżnych, nie występują one w aplikacjach jednowątkowych. Wyścigi obserwujemy zarówno na platformach wieloprocesorowych, jak też jedno-procesorowych (z punktu widzenia aplikacji moment przełączenia procesora jest losowy).

Szybki start

Aby uruchomić przedstawione przykłady, należy mieć dostęp do kompilatora C++ oraz edytora tekstu. Przykłady wykorzystują biblioteki boost (www.boost.org). Aby poprawnie je skompilować, należy dodać odpowiednie zależności wykorzystywane podczas konsolidacji; dla konsolidatora g++ należy dodać opcje: `-lboost_thread -lboost_date_time`; dla konsolidatora Visual Studio (program `link`) biblioteki boost są dodawane automatycznie. Dodatkowo należy używać współbieżnej wersji biblioteki standardowej (dla `g++` opcja `-pthread`, dla `cl` opcja `/MD`). Na wydrukach pominięto dołączanie odpowiednich nagłówków oraz udostępnianie przestrzeni nazw, pełne źródła umieszczono jako materiały pomocnicze.

Blokady

Aplikacje wielowątkowe są trudniejsze do implementacji, ponieważ musimy zapobiegać wyścigom. Analiza wszystkich możliwych, rzeczywistych ciągów instrukcji jest zbyt skomplikowana, dlatego zazwyczaj korzystamy ze specjalnych obiektów, nazywanych obiektami synchronizującymi. Obiekty takie są dostarczane przez system operacyjny. Wykorzystują one instrukcje atomowe, instrukcje, które nie mogą być przerwane w trakcie ich wykonywania oraz wspólną pamięć dla różnych wątków. Za pomocą takich obiektów eliminujemy wyścigi.

Blokady (inna nazwa to muteksy) są jednym z rodzajów obiektów synchronizujących. Zapewniają one wzajemne wykluczanie się wątków (ang. mutual exclusion – wzajemne wykluczanie). Muteksy są dostarczane między innymi przez bibliotekę boost::thread. Służą one do tworzenia sekcji krytycznych, czyli fragmentów kodu, które są wykonywane przez jeden wątek w danym momencie. Obiekt typu `mutex` ma dwa stany: zablokowany (zajęty) i odblokowany (wolny). Metoda `lock` sygnalizuje, że wątek chce wejść do sekcji krytycznej. Jeżeli muteks jest odblokowany (wolny), to metoda ta zmienia stan muteksa – po jej wykonaniu będzie on zablokowany (zajęty), natomiast wątek kontynuuje swoje działanie (nie jest przerywany), czyli wchodzi do sekcji krytycznej. Jeżeli muteks jest zablokowany (zajęty), to metoda `lock` wstrzymuje wykonywanie się wątku (wątek jest przerywany), czeka on na zwolnienie sekcji krytycznej (odblokowanie muteksa). W ten sposób tylko jeden wątek może wykonywać instrukcje wewnątrz sekcji krytycznej. Metoda `unlock` sygnalizuje fakt opuszczania sekcji krytycznej przez wątek. Instrukcja ta nie przerywa działania danego wątku, natomiast bada, czy chociaż jeden wątek oczekuje na zwolnienie danego muteksa. Jeżeli tak, to wykonanie jednego z wątków oczekujących jest wznawiane, wątek ten wchodzi do sekcji krytycznej, stan muteksa się nie zmienia (cały czas jest zajęty). Jeżeli żaden wątek nie oczekuje na zwolnienie muteksa, to jego stan jest zmieniany na odblokowany (wolny).

Sekcja krytyczna jest fragmentem kodu zawartym pomiędzy metodą `lock` i `unlock` dla danej blokady. Ze względu na wzajemne wykluczanie się wątków, wewnątrz sekcji krytycznej nie mogą wystąpić wyścigi. Sekcja krytyczna może zawierać odwołania i modyfikacje zasobu służącego do komunikacji pomiędzy wątkami lub zasobu, o który wątki współzawodniczą.

Biblioteka boost::thread dostarcza obiektów pomocniczych, które zajmują muteks w konstruktorze i zwalniają go w destruktorze, podobnie jak sprytne wskaźniki opisane w SDJ 11/2009 zwalniają obiekt dynamiczny. Tego typu obiekty upraszczają tworzenie sekcji krytycznych, ponieważ nie musimy jawnie zwalniać blokad. Zapobiegają one błędom polegającym na braku wołania metody `unlock` po zakończeniu sekcji krytycznej. Błąd taki łatwo popełnić, gdy kod wewnątrz sekcji krytycznej zgłasza wyjątek.

Przykład wyścigów występujących przy dostępie do tego samego zasobu zawiera Listing 1. Kilka wątków

Listing 1. *Kod zawierający wyścig*

```
struct Counter { // licznik -- demonstruje wyścig
   Counter() : count_(0) { }
   int getC() const { return count_; }
   void setC(int val) { count_ = val; }
private:
   int count_;
};
Counter counter; // obiekt globalny - współdzielony przez wątki

void thread_fun() { // funkcja główna wątku obliczeniowego
   for(int i=0;i<1000000;++i)
      counter.setC( counter.getC() + 1);
}
int main() {
   boost::thread_group thrds;
   for (int i=0; i < 4; ++i)
      thrds.create_thread(&thread_fun);
   thrds.join_all();
   std::cout << "counter: " << counter.getC() << std::endl;
   return 0;
}
```

Zarządzanie wątkami za pomocą biblioteki boost::thread

Biblioteka boost::thread zawiera typy, które pozwalają uruchamiać wątki, zarządzać nimi i organizować komunikację. Sposoby zarządzania wątkami za pomocą tej biblioteki zostały opisane w poprzednim numerze SDJ. Listing 1 zawiera przykład użycia tej biblioteki, który stanowi krótkie przypomnienie. Tworzone są cztery wątki użytkownika (metoda `create _ thread`), każdy z nich wykonuje tę samą funkcję `thread _ fun`. Wątek główny czeka na zakończenie tych wątków, woła metodę `join _ all`.

jednocześnie modyfikuje obiekt globalny `counter`. Ponieważ operacje modyfikacji licznika nie są atomowe (specjalnie zostały tutaj rozbite na metody `getC` oraz `setC`), to zdarzają się sytuacje, gdy wątek A odczyta licznik, następnie wątek B odczyta i zmodyfikuje licznik, później wątek A zapisze licznik, nie uwzględniając tego, że wartość licznika została zmieniona przez wątek B. Liczba, która jest drukowana na koniec działania przedstawionego programu, może być mniejsza od 4000000.

Modyfikacja licznika wewnątrz sekcji krytycznej rozwiązuje problem. Poprawiony kod funkcji `thread_fun` obrazuje Listing 2. Aplikacja działa nieco wolniej, wątki muszą czekać na dostęp do licznika, ale jego stan jest zawsze poprawny. Oprócz licznika musimy dostarczyć blokadę, na Listingu 2 jest to obiekt globalny o nazwie `mutex`.

Zakleszczenia

Jeżeli blokada jest zajęta, to każdy wątek, który będzie chciał wejść do sekcji krytycznej, będzie czekał na jej zwolnienie, dlatego sekcje krytyczne powinny być krótkie. Błędne zarządzanie blokadami może prowadzić do zakleszczeń (ang. deadlock), to znaczy sytuacji, gdy istnieje zbiór wątków, w którym każdy z wątków czeka na jakiś inny (jest blokowany), więc żaden z nich nie może dalej pracować.

Przykład zakleszczenia dla dwóch wątków używających muteksów A i B pokazano na Rysunku 2. Ponieważ wątki zajmują sekcje krytyczne w innej kolejności, może się zdarzyć sytuacja, gdy wątek 1 blokuje muteks A, wątek 2 blokuje muteks B i czeka na odblokowanie muteksa A (który został zablokowany przez wątek 1), następnie wątek 1 próbuje blokować muteks B i czeka na jego odblokowanie przez wątek 2. Żaden z wątków nie może kontynuować swojej pracy.

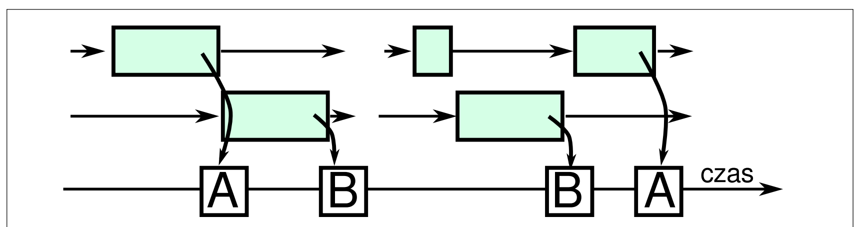
Podobny efekt obserwujemy w kodzie, który nie zwalnia blokady - próba dwukrotnego wołania metody `lock` przez ten sam wątek dla tej samej blokady powoduje zablokowanie wątku.

Aby unikać zakleszczeń, zawsze zwalniajmy blokady. Jeżeli wykorzystujemy wiele blokad, należy zwrócić uwagę na możliwość zakleszczeń i zajmować je w tej samej kolejności w każdym z wątków. Dwukrotne wołanie metody `lock` dla tej samej blokady w tym samym wątku może się zdarzyć, gdy sekcja krytyczna jest częścią algorytmu rekurencyjnego. W takim przypadku warto stosować specjalny typ blokady, `recursive_mutex`. Blokada typu `recursive_mutex` bada identyfikator wątku, który próbuje ją zająć – jeżeli jest to wątek aktualnie znajdujący się w sekcji krytycznej, nie jest on blokowany.

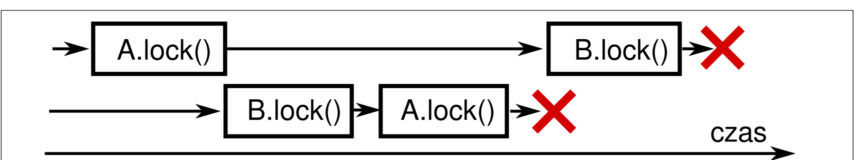
Biblioteka boost::thread dostarcza trzech rodzajów blokad opisanych w Tabeli 1, przy czym każda z blokad ma swój odpowiednik, dla którego można wielokrotnie wołać metodę `lock` w tym samym wątku, o nazwach rozpoczynających się od `recursive` (`recursive_mutex`, `recursive_try_mutex`, `recursive_timed_mutex`).

Zmienne warunkowe

Biblioteka boost::thread dostarcza zmienne warunkowe (ang. *condition variables*), które są innym (niż muteksy) rodzajem mechanizmu synchronizującego. Zmienna warunkowa jest powiązana z pewną blokadą (pewnym muteksem) i służy do powiadamiania wątków o zdarzeniu. Wątek, który czeka na zdarzenie, woła metodę `wait` wewnątrz sekcji krytycznej realizowanej za pomocą blokady powiązanej z daną zmienną.



Rysunek 1. *Przykład wyścigu, dwa wątki piszą do tego samego obiektu*



Rysunek 2. *Przykład zakleszczenia; wątek 1 zajmuje muteks A, wątek 2 zajmuje muteks B i czeka na muteks A, wątek 1 czeka na muteks B*

Listing 2. *Funkcja główna wątku obliczeniowego nie zawierająca wyścigu*

```
void thread_fun() {
   for(int i=0;i<1000000;++i) {
      boost::mutex::scoped_lock scoped_lock(mutex); //konstruktor woła metodę lock
      counter.setC( counter.getC() + 1);
   } //destruktor obiektu scoped_lock woła metodę unlock
}
```

Metoda `wait` zawiesza wykonanie wątku oraz zwalnia blokadę - inne wątki będą mogły wchodzić do sekcji krytycznej. Wątek, który chce powiadomić wątki oczekujące na zdarzenie, tzn. wątki, które wywołały metodę `wait` i zostały zawieszone, woła metodę `notify_one` albo `notify_all`. Metoda ta wznawia wątki oczekujące (jeden lub wszystkie). Wznowienie oznacza próbę zajęcia blokady powiązanej z daną zmienną warunkową. Jeżeli wątek zostanie „wpuszczony" do sekcji krytycznej, to sterowanie będzie przekazane do instrukcji następnej po `wait`.

Zmienne warunkowe znacznie ułatwiają implementację komunikacji pomiędzy różnymi wątkami, gdy obiekty tworzone przez dany wątek są danymi wejściowymi dla innych wątków. Listing 3 zawiera przykład wykorzystujący zmienne warunkowe. Są tam pokazane dwie funkcje, które są wykonywane w różnych wątkach: funkcja `produce_data`, uruchamiana w wątkach nazywanych producentami, oraz funkcja `consume_data`, uruchamiana w wątkach nazywanych konsumentami. Producenci generują dane, konsumenci wykorzystują te dane. Dostęp do danych jest synchronizowany za pomocą blokady `mut`. W przedstawionym przykładzie dane są reprezentowane przez flagę `ready`. Producent zajmuje sekcję krytyczną i dostarcza daną. Po wykonaniu tej akcji powiadamia wszystkich oczekujących konsumentów o tym fakcie, wykorzystując zmienną warunkową `cond`. Konsument czeka na zdarzenie o dostępności danych, które są przetwarzane wewnątrz sekcji krytycznej.

Singleton

Singleton ogranicza możliwość tworzenia obiektów danej klasy do jednej instancji, dostarczając globalny dostęp do tego obiektu. Potrzeba taka występuje, gdy klasa reprezentuje typ, który opisuje byt występujący w pojedynczym egzemplarzu (na przykład klawiaturę) albo taki, który powinien być wspólny dla całej aplikacji (na przykład fabryka obiektów danego typu). Do tego celu może być użyty obiekt zdefiniowany w zasięgu globalnym lub obiekt, który jest składową statyczną klasy. Rozwiązanie to prowadzi do błędów związanych z kolejnością inicjacji takich obiektów. Są one inicjowane przed wołaniem funkcji `main`. Podczas kompilacji rejestruje się definicje takich obiektów, kolejność tej rejestracji jest kolejnością ich inicjowania (tworzenia). Obiekty globalne lub składowe statyczne zazwyczaj są zdefiniowane w różnych plikach. Nie mamy możliwości ustalenia kolejności kompilacji poszczególnych plików, więc musimy przyjąć, że kolejność inicjacji obiektów globalnych i statycznych jest losowa. Jeżeli obiekty globalne korzystają z innych obiektów globalnych, kolejność ich inicjacji jest ważna, używanie niezainicjowanych obiektów powoduje błędne (niezdefiniowane) działanie programu. Przedstawiony wyżej błąd jest trudny do wykrycia, ponieważ może czasami nie wystąpić (jeżeli przez przypadek kolejność inicjacji obiektów będzie właściwa).

Przedstawione dalej rozwiązanie, singleton, pozwala rozwiązać problem inicjacji, stosując pośrednie odwołanie do obiektu globalnego. Dodatkowo umieszczamy w danej klasie informację, że można utworzyć tylko jeden obiekt tej klasy.

Listing 4 zawiera przykład singletona. Nie można utworzyć obiektu typu `Singleton` poza metodami klasy, ponieważ wszystkie konstruktory są prywatne. Metoda `getInstance` dba o to, aby był tworzony tylko jeden obiekt. Jest on tworzony przy pierwszym wołaniu tej metody. W każdym kolejnym żądaniu dostępu do obiektu jest zwracany wskaźnik do obiektu utworzonego wcześniej. Problem nieznanej (losowej) kolejności inicjacji obiektów utworzonych w pamięci statycznej nie występuje, obiekt zarządzany przez singleton jest tworzony przy pierwszym żądaniu dostępu do niego, nie ma możliwości odwołania się do obiektu niezainicjowanego.

Kosztem stosowania singletonów w porównaniu z obiektem globalnym jest przechowywanie dodatkowego uchwytu (wskaźnika) do obiektu. Dodatkowo, podczas dostępu wykonuje się instrukcję warunkową, która bada, czy obiekt istnieje.

Wzorzec podwójnego sprawdzania

Singleton przedstawiony na Listingu 4 może powodować wyścig. Jeżeli obiekt nie został utworzony, a dwa wątki niezależnie wołają metodę `getInstance`, może

Tabela 1. *Blokady dostarczane przez boost::thread*

Nazwa	Opis
mutex	typ opisany w tekście
try_mutex	wykorzystuje metody try_lock i unlock. Metoda try_lock działa inaczej niż lock. Jeżeli muteks jest wolny (odblokowany), to metoda ta zajmuje go i zwraca wartość true. Jeżeli muteks jest zajęty (zablokowany). to metoda nie przerywa wykonywania wątku wołającego ją, ale zwraca wartość false. Wątki nigdy nie oczekują na wejście do sekcji krytycznej.
timed_mutex	wykorzystuje metody timed_lock i unlock. Przy wejściu do sekcji, jeżeli muteks jest zablokowany, to wątek czeka pewien czas (czas ten jest argumentem metody). Jeżeli w tym czasie wątek zostanie „wpuszczony" do sekcji krytycznej, to metoda ta zwraca true. Po przekroczeniu określonego czasu metoda zwraca false i wątek może kontynuować pracę, sekcja nie jest zajmowana.

zdarzyć się następujący scenariusz: wątek A bada warunek (instrukcja `if` w metodzie `getInstance`), ponieważ obiekt nie istnieje, warunek jest prawdziwy. Następnie sterowanie przejmuje wątek B, który również widzi brak obiektu zarządzanego przez singleton, więc go tworzy i zwraca. Później, gdy wątek A będzie kontynuować swoje wykonanie, utworzony zostanie kolejny obiekt singletona. Wystąpił wyścig – obiekt `pInstance_` został utworzony dwukrotnie.

Objęcie całej metody `getInstance` sekcją krytyczną rozwiązuje problem, ale zmniejsza wydajność. Za każdym razem, gdy będziemy wołali tę metodę, będzie blokowany i odblokowywany muteks, którego jedynym zadaniem jest poprawne tworzenie obiektu. Zajmowanie sekcji krytycznej, gdy obiekt istnieje, jest bezcelowe. Lepszym rozwiązaniem jest propozycja przedstawiona na Listingu 5, która zachowuje się poprawnie przy tworzeniu obiektu, natomiast przy dostępie nie wymaga używania sekcji krytycznej.

Pokazany sposób jest nazywany wzorcem podwójnego sprawdzania. Dwukrotnie badamy fakt istnienia obiektu. W sytuacji, gdy obiekt nie istnieje (jeżeli wskaźnik `pInstance` jest pusty), wykorzystywana jest sekcja krytyczna, wewnątrz której tworzony jest obiekt. Niezbędne jest powtórne sprawdzenie warunku, aby nie wystąpił wyścig opisany wyżej. Gdy obiekt istnieje, sekcja krytyczna nie jest używana.

Poprawność wzorca podwójnego sprawdzania nie jest gwarantowana przez standard języka. Wzorzec ten wymaga, aby linia `pInstance_ = new Singleton;` (patrz Listing 5) była wykonywana w następującej kolejności: 1) przydział pamięci dla obiektu, 2) wołanie konstruktora, 3) przypisanie adresu do obiektu `pInstance_`. Standard dopuszcza możliwość wykonania tej linii w innej kolejności, np.: przydział pamięci, przypisanie wskaźnika, a później wołanie konstruktora. Jeżeli taka kolejność byłaby wygenerowana, warunek badający istnienie obiektu (niepusty wskaźnik) byłby prawdziwy dla

Listing 3. *Przykład wykorzystujący zmienne warunkowe*

```
boost::condition_variable cond;
boost::mutex mut;
bool ready;

void consume_data() { // funkcja główna wątku odbierającego dane
   mutex::scoped_lock lock(mut); // początek sekcji krytycznej
   while( ! ready ) {
      cond.wait( lock ); // czeka na powiadomienie, zwalnia muteks
   }
   // tutaj flaga ready == true i jest w sekcji krytycznej
   // tutaj przetwarza dane
   ready = false; // ustawia flag na nieaktywną
}
void produce_data() { // funkcja główna wątku dostarczającego dane
   // przygotowuje dane
   { // sekcja krytyczna
      mutex::scoped_lock lock(mut); // początek sekcji krytycznej
      ready = true; // ustawia flagę
   }
   cond.notify_all(); // powiadamia wszystkich
}
```

Listing 4. *Singleton*

```
class Singleton { //przykładowy singleton
public:
   static Singleton* getInstance() { //dostęp do obiektu
      if(!pInstance_) //zapobiega tworzeniu wielu kopii
         pInstance_ = new Singleton;
      return pInstance_;
   }
   //pozostałe metody
private:
   Singleton(); //prywatny konstruktor
   Singleton(const Singleton&); //prywatny konstruktor kopiujący
   static Singleton* pInstance_; //deklaracja składowej globalnej
};
```

Listing 5. *Singleton stosujący podwójne sprawdzanie*

```
Singleton& Singleton::getInstance() {
   if(!pInstance_) {
      Lock guard(mutex_); //tworzy sekcję krytyczną
      if(!pInstance_)
         pInstance_ = new Singleton;
    }
   return pInstance_;
}
```

Więcej w książce

Zagadnienia dotyczące współcześnie stosowanych, w języku C++, technik, wzorce projektowe, programowanie generyczne, prawidłowe zarządzanie zasobami przy stosowaniu wyjątków, programowanie wielowątkowe, ilustrowane przykładami stosowanymi w bibliotece standardowej i bibliotekach boost, zostały opisane w książce Robert Nowak, Andrzej Pająk „Język C++: mechanizmy, wzorce, biblioteki", która ukaże się niebawem.

W Sieci

- M. Ben-Ari. *Podstawy programowania współbieżnego i rozproszonego*, WNT, 1996;
- *http://www.boost.org* – dokumentacja biblioteki boost.
- *www.cs.wustl.edu/~schmidt/PDF/DC-Locking.pdf* – artykuł Douglasa Shmidta i Toma Harrisona na temat wzorca podwójnego sprawdzania.

obiektów, które nie zostały do końca utworzone (konstruktor się nie zakończył). Pomimo teoretycznej niepoprawności wzorca podwójnego sprawdzania dla singletonów, nie zaobserwowano niepoprawnego zachowania się przedstawionego kodu na znanych platformach (analizowano assembler, kolejność akcji była zgodna z przedstawionym wyżej opisem). Z drugiej strony singletony warto inicjować przez pojedynczy wątek, przy starcie programu, a wtedy problem zapobiegania wyścigom podczas inicjacji znika.

Podsumowanie

Przedstawione techniki omawiają sposoby organizowania komunikacji pomiędzy różnymi wątkami. Najlepiej użyć wspólnych obiektów, ale wtedy trzeba eliminować wyścigi. W większości przedstawionych przykładów używamy obiektów synchronizacyjnych (muteksy, zmienne warunkowe) dostarczanych przez bibliotekę boost::thread. Warto minimalizować liczbę takich obiektów, ponieważ np. zajmowanie i zwalnianie blokady jest czasochłonne. Jednym ze sposobów na zmniejszenie ilości takich obiektów jest stosowanie wzorca podwójnego sprawdzania, innym sposobem, przydatnym szczególnie przy obsłudze urządzeń wejścia-wyjścia jest obsługa zdarzeń.

ROBERT NOWAK

Adiunkt w Zakładzie Sztucznej Inteligencji Instytutu Systemów Elektronicznych Politechniki Warszawskiej, zainteresowany tworzeniem aplikacji bioinformatycznych oraz zarządzania ryzykiem. Programuje w C++ od ponad 10 lat.

Kontakt z autorem:rno@o2.pl

REKLAMA

Wybierz swój nowy system, poznaj Linuksa krok po kroku!

Twoje pierwsze źródło wiedzy o Linuksie

w wortalu:

– recenzje dystrybucji

– porównanie funkcji

– opis instalacji

– i dużo więcej

ponad ćwierć miliona odwiedzin miesięcznie!

jakilinux.org

kontakt / reklama

borys.musielak@gmail.com