Asynchroniczna obsługa urządzeń wejścia – wyjścia

Biblioteka boost::asio

Urządzenia wejścia – wyjścia działają znacznie wolniej niż procesor, dlatego w czasie oczekiwania na odpowiedź urządzenia warto go zwalniać. Współczesne systemy operacyjne dostarczają udogodnień, które pozwalają to osiągnąć bez angażowania niezależnych wątków.

Dowiesz się:

- Do czego służy biblioteka boost::asio;
- Jak generować zdarzenia związane z zegarem;
- Jak obsługiwać gniazda.

Powinieneś wiedzieć:

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

Wstęp

Urządzenia wejścia i wyjścia pracują znacznie wolniej niż procesor, dlatego warto stosować mechanizmy, które pozwalają wykorzystywać wolną moc obliczeniową. Urządzenia możemy obsługiwać synchronicznie bądź asynchronicznie, w jednym i drugim przypadku możemy wykonywać pewne akcje podczas oczekiwania na odpowiedź urządzenia.

Synchroniczna obsługa wejścia lub wyjścia jest prostsza niż asynchroniczna. Polega ona na blokowaniu bieżącego strumienia instrukcji na czas obsługi urządzenia. Mechanizm komunikacji z urządzeniem jest następujący: wysyłamy żądanie (np. wołamy metodę), następuje jego obsługa, która blokuje strumień instrukcji, zaś gdy urządzenie da odpowiedź, to sterowanie jest nam zwracane, następuje powrót z metody. Ze względu na wspomnianą wcześniej różnicę w szybkości działania urządzeń oraz procesora, aby wykorzystać dostępną moc obliczeniową, należy używać dodatkowych wątków do obsługi urządzeń. Wątek taki wysyła żądanie do urządzenia i czeka na jego odpowiedź, udostępniając w tym samym czasie procesor innym wątkom.

Rozwiązanie powyższe ma pewne wady: wymaga stosowania dużej ilości wątków, należy zapewnić poprawny dostęp do obiektów służących do wymiany danych. Często jesteśmy zmuszeni stosować blokady (lub inne mechanizmy służące do synchronizacji, patrz SDJ 11/2010), a to jest kosztowne.

Asynchroniczna obsługa wejścia – wyjścia przebiega według następującego scenariusza: aplikacja rejestruje żądania obsługi wejścia-wyjścia w obiekcie, który przekazuje żądanie do systemu operacyjnego; rejestracja żądania rozpoczyna obsługę urządzenia, zaś sterowanie jest zwracane do aplikacji (nie czeka się na odpowiedź). System operacyjny wykonuje operację wejścia-wyjścia, wypełnia odpowiedni bufor, a następnie woła funkcję użytkownika, dostarczoną podczas rejestracji żądania. Przy obsłudze asynchronicznej nie ma potrzeby stosowania wielu wątków, co pozwala unikać kosztów związanych z przełączaniem wątków oraz zapewnienia poprawn-

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 -lboost_system`; 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.

nego dostępu do obiektów przechowujących dane. Zazwyczaj można jednocześnie rejestrować (rozpoczynać) wiele operacji wejścia – wyjścia. Będą one obsługiwane wtedy, gdy otrzymamy odpowiedź z urządzenia.

Biblioteka boost::asio

Biblioteka `boost::asio` dostarcza udogodnień umożliwiających tworzenie przenośnego kodu obsługującego urządzenia wejścia – wyjścia. Dostarczono możliwość obsługi:

- gniazd TCP i UDP (w tym dostarczanie strumieni związanych z gniazdami, a także strumieni kodowanych SSL),
- zegarów (generują zdarzenia po pewnym czasie),
- portów szeregowych.

Biblioteka wykorzystuje wątki `boost::thread` (omówione w SDJ 10/2010 oraz SDJ 11/2010); została ona przetestowana na wielu platformach (m.in. Linux, QNX, Solaris, Mac OS X, Windows: 95, NT, XP, Vista). Wymienione urządzenia możemy obsługiwać synchronicznie bądź asynchronicznie, biblioteka dostarcza udogodnień dla obu technik.

Synchroniczna obsługa operacji wejścia-wyjścia, przy użyciu biblioteki `boost::asio`, polega na:

- wywołaniu operacji dla obiektu wejścia-wyjścia (np. wołanie metody `wait` dla zegara);
- metoda ta przekazuje odpowiednie żądanie do obiektu `io_service` (zawsze przynajmniej jeden taki obiekt powinien istnieć);
- `io_service` przekazuje żądanie do systemu operacyjnego i czeka na wynik;
- system operacyjny realizuje żądanie i zwraca wynik do obiektu `io_service`;
- wynik ten jest przekazywany do odpowiedniego obiektu wejścia-wyjścia;
- obiekt wejścia-wyjścia kończy działanie metody, zwracając odpowiednie dane.

Scenariusz wykonywany podczas asynchronicznej obsługi urządzenia jest następujący:

- użytkownik woła operację dla obiektu wejścia-wyjścia (np. woła metodę `async_wait`), przekazując uchwyt do funkcji (lub obiektu funkcyjnego), która ma być wołana, gdy operacja zostanie wykonana;
- obiekt wejścia-wyjścia przekazuje żądanie (oraz uchwyt do funkcji) do obiektu `io_service`;
- obiekt `io_service` przekazuje żądanie rozpoczęcia operacji do systemu operacyjnego. Metoda inicjacji żądania się kończy, użytkownik może wykonywać inne operacje;
- gdy żądanie zostanie zrealizowane przez system operacyjny, jego wynik zostaje dodany do kolejki, którą zawiera obiekt `io_service`;
- użytkownik może żądać obsługi zarejestrowanych operacji asynchronicznych, wołając metodę `run` dla obiektu `io_service`. Metoda ta pobiera wynik danej operacji z kolejki, a następnie woła odpowiednią funkcję, która została przekazana przy rejestracji żądania. Jeżeli istnieją rozpoczęte operacje wejścia – wyjścia, to metoda ta blokuje wątek ją wołający, dopóki nie pojawi się wynik jednego z zarejestrowanych zdarzeń. Metoda ta kończy się, gdy zostaną obsłużone wszystkie żądania.

Generowanie zdarzeń związanych z zegarem

Jednym z urządzeń obsługiwanych przez opisywaną tutaj bibliotekę jest zegar systemowy (lub inne urządzenie odmierzające czas). Obiekt typu `deadline_timer` ge-

Listing 1. *Przykład wykorzystania urządzenia odmierzającego czas*

```
// Funkcja wołana, gdy zajdzie odpowiednie zdarzenie
void event(const boost::system::error_code&) {
   cout << "timer event" << endl;
}

int main() {
   boost::asio::io_service io; // obiekt obsługujący zdarzenia
   boost::asio::deadline_timer t1(io, boost::posix_time::seconds(4)); // generuje zdarzenie po 4 sek
   t1.wait(); // czeka na zdarzenie (obsługa synchroniczna)
   cout << "timer event" << endl;
   boost::asio::deadline_timer t2(io, boost::posix_time::seconds(4)); //generuje zdarzenie po 4 sek
   t2.async_wait(event); // obsługa asynchroniczna - rejestracja zdarzenia
   io.run();// obsługa zarejestrowanych zdarzeń asynchronicznych
   return 0;
}
```

neruje zdarzenie po pewnym czasie. Przykład wykorzystania pokazano na Listingu 1.

W przedstawionym przykładzie tworzone są dwa obiekty reprezentujące urządzenie odmierzające czas (obiekty `t1` oraz `t2`). Oba generują zdarzenie po 4 sekundach. Zdarzenia związane z tymi obiektami są obsługiwane przez wątek główny.

Zegar `t1` jest obsługiwany synchronicznie, wołamy metodę `wait`, więc wątek główny czeka na zdarzenie związane z tym zegarem. Zegar `t2` jest obsługiwany asynchronicznie (metoda `async_wait`). Obiekt `io_service`, wykorzystywany przy synchronicznej i asynchronicznej obsłudze urządzeń, przechowuje uchwyty, które są wołane po uzyskaniu odpowiedzi z urządzenia. Obiekt ten udostępnia metodę `run`, która zawiesza bieżący wątek do chwili obsługi wszystkich zarejestrowanych żądań. Jeżeli którekolwiek z zarejestrowanych żądań zostanie zrealizowane, to jest wołana odpowiednia funkcja (w wątku, który wywołał metodę `run`). W przedstawionym przykładzie rejestrujemy tylko jedno żądanie asynchroniczne, obsługę zegara `t2`, więc funkcja `main` zakończy się, po obsłużeniu zdarzenia wysyłanego przez ten zegar.

Obsługa protokołów sieciowych

Głównym przeznaczeniem biblioteki `boost::asio` jest dostarczenie przenośnych mechanizmów obsługi protokołów sieciowych. Schemat postępowania przy tworzeniu gniazd i przesyłaniu danych jest podobny do ogólnego scenariusza obsługi urządzenia synchronicznego lub asynchronicznego. Użytkownik inicjuje żądanie i albo czeka na wynik (wołanie synchroniczne), albo przekazuje odpowiedni obiekt funkcyjny (wołanie asynchroniczne). Na Listingu 2 pokazano przykładowy program, który pobiera stronę HTML z serwera o adresie `127.0.0.1`.

Aby wysłać lub odebrać dane, trzeba utworzyć obiekt wejścia-wyjścia reprezentujący gniazdo. Następnie nawiązujemy połączenie, określając punkt docelowy, na Listingu 2 posłużyliśmy się adresem IP i numerem portu. Biblioteka obsługuje adresy cztero- i sześcio-bajtowe, pozwala ona także określić adres i port na podstawie nazwy domeny i nazwy usługi. Obiektem wejścia-wyjścia odpowiedzialnym za transmisję, który został przedstawiony w przykładzie, jest obiekt `socket`, reprezentuje on połączenie TCP. Dla tego obiektu wołana jest metoda `connect`, która zleca synchroniczne nawiązanie połączenia. Metoda ta kończy się, gdy połączenie zostanie nawiązane lub wystąpi błąd. Jeżeli chcielibyśmy żądanie połączenia realizować asynchronicznie, należałoby wołać metodę `async_connect` dla odpowiedniego gniazda.

Listing 2. *Program pobierający stron HTML z serwera localhost*

```
std::size_t completion(const boost::system::error_code& error, std::size_t bytes_transferred) {
   return ! error; // kończy gdy dowolny błąd
}
int main()
{
   io_service io_service; // obsługuje żądania wejścia-wyjścia

   ip::address address = ip::address::from_string("127.0.0.1");
   ip::tcp::endpoint endpoint(address, 80);
   ip::tcp::socket socket(io_service);
   socket.connect(endpoint); // synchroniczne nawiązanie połączenia

   streambuf request; // przechowuje żądanie wysyłane do serwera www
   std::ostream request_stream(&request);
   request_stream   << "GET index.html HTTP/1.0\r\n"
                    << "Host: localhost\r\n"
                    << "Accept: */*\r\n"
                    << "Connection: close\r\n\r\n";
   write(socket, request); // synchronicznie wysyła żądanie
   streambuf response;
   boost::system::error_code ec;
   read(socket, response, completion, ec); // odczytuje odpowiedź
   std::cout << &response;
   return 0;
}
```

Gniazda dostarczają metody: `read_some`, `read_async_some`, `write_some`, `write_async_some`, które odczytują lub zapisują, synchronicznie lub asynchronicznie pewną porcję danych. Dodatkowo biblioteka `boost::asio` dostarcza funkcje pomocnicze: `read`, `read_async`, `write`, `write_async`, które zapisują lub odczytują strumień danych, wołając (być może wielokrotnie) odpowiednie metody (np. `read_some`) dla dostarczonego obiektu wejścia-wyjścia. Wykorzystując te funkcje użytkownik może wysyłać obiekty o dowolnej wielkości, podział na odpowiednie porcje danych będzie realizowany przez te funkcje. Analizowany przykład wykorzystuje funkcję `write` do synchronicznego wysłania żądania do serwera www. Żądanie jest ciągiem znaków przechowywanym przez strumień `request`.

Odczyt danych z gniazda, w przedstawionym przykładzie, jest także synchroniczny (podobnie jak nawiązywanie połączenia), używamy funkcji `read`. Funkcja ta czyta kolejne paczki danych, badając warunek końca transmisji.

Wykorzystanie puli wątków

Jeżeli tylko jeden wątek woła metodę `io_service::run`, to mamy gwarancję obsługi asynchronicznych zdarzeń wejścia-wyjścia w jednym wątku. Nie musimy wtedy stosować blokad lub innych mechanizmów synchronizacji, które są niezbędne przy wielowątkowej obsłudze zdarzeń. Jeżeli jeden wątek nie jest w stanie obsługiwać wszystkich żądań wejścia – wyjścia (na przykład w aplikacji serwera, który musi obsługiwać wiele połączeń sieciowych równocześnie) to można obsługę zleceń realizować w kilku wątkach, każdy taki wątek woła metodę `run`. Przy takim podejściu należy synchronizować dostęp do wspólnych zasobów wykorzystywanych przy obsłudze różnych żądań wejścia – wyjścia, ponieważ funkcje obsługi żądania asynchronicznego mogą być uruchamiane jednocześnie w różnych wątkach.

Podsumowanie

Przedstawione przykłady pokazują sposoby obsługi urządzeń wejścia – wyjścia przy pomocy biblioteki `boost::asio`. Biblioteka ta dostarcza udogodnień, które pozwalają w podobny sposób obsługiwać żądania synchroniczne i asynchroniczne. Asynchroniczna metoda obsługi urządzeń posiada wiele zalet w porównaniu z metodą synchroniczną. Ponieważ większość systemów operacyjnych udostępnia taki rodzaj współpracy z urządzeniami, warto go wykorzystywać.

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

W Sieci

- *http://www.boost.org* – dokumentacja bibliotek boost.
- *http://www.staff.amu.edu.pl/~psi/informatyka/tcpip/index.htm* – opis protokołu TCP/IP.

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 w grudniu 2010.

Reklama

TTS Company

Największy wybór oprogramowania w Polsce !

... w ofercie produkty ponad 300 producentów ...

www.OprogramowanieKomputerowe.pl

Microsoft GOLD CERTIFIED Partner · EMBARCADERO TECHNOLOGIES · intel Software · SPARX SYSTEMS · ALTOVA · SmartBear SOFTWARE · IDM · FLEXERA SOFTWARE

TTS Company Sp. z o.o. ul.Słowicza 53 02-170 Warszawa tel./fax: (022) 868 40 42 e-mail: tts@tts.com.pl www.tts.com.pl