Programowanie Aplikacji Sieciowych - Laboratorium 12

Wieloklientowy serwer Do tej pory wszystkie serwery napisane na zajęciach mogły, wykorzystując gniazda blokujące, obsłużyć jednego klienta w tym samym czasie (były jednowątkowe, obsługując jednego klienta nie akceptowały połączeń od innych klientów). Obsługę kilku klientów w tym samym czasie można zrealizować za pomocą jednej z 2 metod:

- Wątki
- Gniazda nieblokujące i monitorwanie zdarzeń

Wątki określane są jako wydzielone sekwencje przewidzianych do wykonania instrukcji, które są realizowane w ramach jednego procesu (programu). Każdy wątek ma swój własny stos, zestaw rejestrów, licznik programowy, indywidualne dane, zmienne lokalne, i informację o stanie. Wszystkie wątki danego procesu mają jednak tę samą przestrzeń adresową, ogólną obsługę sygnałów, pamięć wirtualną, dane oraz wejście-wyjście. W ramach procesu wielowątkowego każdy wątek wykonuje się oddzielnie i asynchronicznie. Każdy proces zawiera przynajmniej jeden główny wątek początkowy, tworzony przez system operacyjny w momencie stworzenia procesu (programu), stąd możemy powiedzieć, że wszystkie napisane do tej pory serwery, były serwerami jednowątkowymi. Wraz z uruchomieniem programu jeden wątek, zwany wątkiem głównym, startuje natychmiast. Wątek główny jest ważny, ponieważ jest wątkiem mogącym zapoczątkowywać inne wątki, zwane wątkami potomnym, często także musi być ostatnim wątkiem kończącym wykonanie programu.

Jak wykonywany jest program wielowątkowy, gdy mamy do dyspozycji:

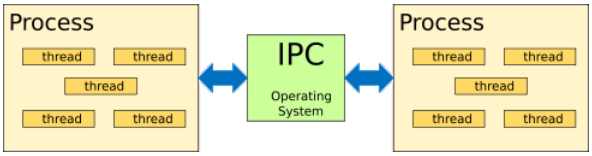
- **1 CPU** (szybkie przełączanie zadań - iluzja równoległości; emulacja współbieżności)

- **2 CPU** (sytuacja idealna; zasada obowiązuje dla 2 i więcej CPU)

- **2 CPU** (sytuacja rzeczywista; zasada obowiązuje dla 2 i więcej CPU)

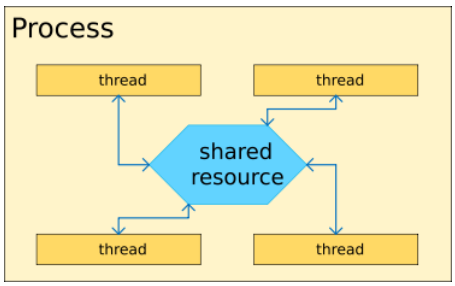
Czym jest proces?

- egzemplarz wykonywanego programu
- kontener wątków w chronionej przestrzeni adresowej, system operacyjny przydziela procesowi (alokuje) odpowiednie zasoby
- za zarządzanie procesami odpowiada jądro systemu operacyjnego
- system operacyjny zarządza priorytetami procesów



Czym jest wątek?

- niezależny ciąg instrukcji wykonywany współbieżnie w ramach jednego procesu
- wszystkie wątki działające w danym procesie współdzielą przestrzeń adresową oraz zasoby systemowe
- zasobem, który wątki nie współdzielą jest pamięć stosu (stack). Każdy utworzony wątek posiada własny stos, na którym alokowane są lokalne zmienne funkcji wywołanej w tym wątku



Wielowątkowe serwery Serwery wielowątkowe są w stanie obsługiwać równolegle wiele połączeń od wielu klientów. W wątku głównym serwer powinien nasłuchiwać na określonym porcie o ustalonym numerze. W momencie, kiedy nadchodzi żądanie otwarcia nowego połączenia, serwer powinien je obsłużyć w oddzielnym, nowym wątku. Ogólny schemat działania serwera wielowątkowego jest następujący:

- Tworzymy gniazdo serwera
- Przypisujemy do gniazda adres IP oraz numer portu, na którym serwer będzie nasłuchiwał na przychodzące połączenia od klientów
- Uruchamiamy pętlę nieskończoną, w której akceptujemy połączenia od klienta, i w przypadku, gdy klient podłączy się do serwera, tworzymy nowy wątek do obsługi klienta

Ponieważ w pętli tworzymy nowy wątek dla każdego nowego, nadchodzącego klienckiego połączenia, serwer może jednocześnie cały czas obsługując już podłączonych do niego klientów, oczekiwać na i akceptować nowe połączenia.

Obsługa wątków Do obsługi wątków:

- W języku Python wykorzystamy moduł `threading`
- W językach C/C++ wykorzystamy wątki POSIX, `#include <pthread.h>`
- W języku Java wykorzystamy klasę `Thread`

Poniższe przykłady prezentują ogólny schemat wielowątkowych tworzenia serwerów TCP w wybranych językach programowania wraz z krótkim opisem.

Python: Schemat obsługi wielu klientów za pomocą wątków:

```
#!/usr/bin/env python
import socket, sys, threading

class ClientThread(threading.Thread):
    def __init__(self, connection):
        threading.Thread.__init__(self)
        # ...

    def run(self):
        # obsluga odbierania i wysylania danych

class Server:
    def __init__(self, ip, port):
        # ...

    def run(self):
        try:
            # socket, bind, listen

            while True:
                # accept

                c = ClientThread(connection)
                c.start()

        except socket.error, e:
            # ...

if __name__ == '__main__':
    s = Server('127.0.0.1', 6666)
    s.run()
```

Do obsługi wielu klientów w języku Python wykorzystamy moduł `threading`. Moduł `threading` udostępnia wysokopoziomowy interfejs wątków. Serwer bez wątków może obsługiwać jednocześnie jednego klienta, a reszta musi czekać. W przypadku zastosowania wątków serwer może obsługiwać wielu klientów jednocześnie, co znacznie zwiększa wydajność.

Zaczynamy od klasy obsługującej nowo podłaczonego klienta, `ClientThread`. Klasa `ClientThread` dziedziczy po klasie `Thread` z modułu `threading`, co umożliwia traktowanie instancji klientów jako wątki, oraz wymusza by metoda `__init__` klasy `ClientThread` wywołała odpowiadającą metodę inicjalizującą klasy `Thread`. Za odbieranie/wysyłanie danych z/do serwera odpowiada metoda `run` klasy `ClientThread`, to w niej znajduje się obsługa komunikacji z klientem. Metoda `run` nadpisuje tę z klasy `Thread`, dzięki czemu przy wywołaniu metody `start()` na wątku `ClientThread` wykona ona automatycznie metodę `run`. Zwróć uwagę, że w tej metodzie serwer może wejść w pętlę ciągłego odczytu danych od klienta aż do czasu rozłączenia się (klienta z serwerem).

Klasa serwera reprezentuje działający serwer. W konstruktorze (`__init__`) wykonywana jest inicjalizacja zmiennych, natomiast w metodzie `run` (której nazwa może być dowolna, ponieważ klasa serwera to zwykła klasa, nie dziedzicząca po klasie `Thread`), tworzone jest gniazdo nasłuchujące serwera, jak również akceptowane są połączenia od klientów. Dla każdego nowego połączenia serwer tworzy nowy wątek klienta (obiekt klasy `ClientThread`).

C/C++: Schemat obsługi wielu klientów za pomocą wątków:

```
1  #include <pthread.h>
2
3  /* g++ tcp_multithreaded_server.cpp -o tcp_multithreaded_server -lpthread */
4
5  void *ClientThread(void *args)
6  {
7      int sock = *(int*)args;
8
9      // odbieranie/wysylanie wiadomosci od/do klienta
10 }
11
12 int main(int argc, char **argv)
13 {
14     // utworzenie gniazda, bind, listen
15
16     // akceptowanie klientow - w petli nieskonczonej
17     while(client_fd = accept(server_fd, (struct sockaddr *) &client, &client_len))
18     {
19         // tworzymy nowy watek
20         pthread_t thread;
21
22         // w watku uruchamiana jest obsluga podlaczajacego sie klienta,
23         // zaimplementowana w funkcji ClientThread
24         if(pthread_create(&thread, NULL, ClientThread, (void*) &client_fd) < 0)
25         {
26             std::cout << ("Could not create a new thread socket\n");
27             perror("pthread_create");
28             return 1;
29         }
30     }
31
32     // zamkniecie socketow
33
34     return 0;
35 }
```

Wszystkie programy korzystające z funkcji operujących na wątkach normy POSIX zawierają dyrektywę `#include <pthread.h>`. Kompilując przy użyciu `gcc` czy `g++` programy korzystające z tej biblioteki, należy wymusić jej dołączenie, przez użycie opcji `-lpthread`.

Każdy proces zawiera przynajmniej jeden główny wątek początkowy, tworzony przez system operacyjny w momencie stworzenia procesu (uruchomienia naszego programu). By do procesu dodać nowy wątek należy wywołać funkcję `pthread_create`. Nowo utworzony wątek zaczyna się od wykonania funkcji użytkownika (przekazanej mu przez argument funkcji `pthread_create`, tutaj jest to funkcja o nazwie `ClientThread`). Kiedy wątek jest tworzony zaczyna wykonywać funkcję wątku. Po zakończeniu funkcji kończy istnienie także wątek.

Nagłówek funkcji obsługującej wątak musi spełniać pewne wymagania, których nie możemy zmienić. Funkcje wątków muszą definiować swój typ zwracany jako `void *`, jak również muszą jako pobierać tylko jeden argument, również typu `void *`. Dzięki takiemu podejściu funkcja może przyjmować właściwie dowolny typ argumentu - przykład w linii 9.

Funkcja uruchamiająca nowy wątek, `pthread_create`, pobiera 4 argumenty (linia 24):

1. wskaźnik do zmiennej typu `pthread_t`
2. dodatkowe atrybuty, które w naszym przypadku nie są do niczego potrzebne; stąd drugi argument powinien przyjąć wartść `NULL`
3. wskaźnik do funkcji, w której znajduje się obsługa wątku
4. wskaźnik do argumentu, który jest traktowany jako argument funkcji obsługującej wątek

Java: Schemat obsługi wielu klientów za pomocą wątków:

```
class ClientThread extends Thread {

    private Socket clientSocket;

    public ClientThread(Socket client) {
        this.clientSocket = client;
    }

    @Override
    public void run() {

        try {

            // wysylanie/odbieranie danych do/od klienta

            clientSocket.close();

        } catch (IOException ex) { }
    }
}

class Server {

    private Socket client_connection = null;
    private ServerSocket server = null;
    private int port = 0;

    public Server(int port) {
        this.port = port;
    }

    public void start() {
        try {

            this.server = new ServerSocket(this.port);

            while (true) {

                client_connection = server.accept();

                ClientThread c = new ClientThread(client_connection);

                c.start();
            }

        } catch (IOException ex) { }
    }
}

public class tcp_multithreaded_server {

    public static void main(String[] args) {

        Server s = new Server(6666);
        s.start();
    }
}
```

W celu utworzenia wątku w Javie mamy dwie podstawowe opcje:

1. dziedziczenie po klasie `Thread`
2. implementacja interfejsu `Runnable`

Różnice:

- Implemetacja interfejsu `Runnable`, gdy klasa już dziedziczy po innej klasie
- Dziedziczenie po klasie `Thread`, jeżeli chcemy zmodyfikować jej metody

W obu przypadkach, metoda `run()` określa co ma robić wątek. Właśnie w metodzie `run()` zapisujemy kod, który będzie wykonywany jako wątek (równolegle z innymi wątkami programu). Wewnątrz jej ciała można wywoływać inne metody, tworzyć obiekty klas. Tu wątek zaczyna się i kończy się. Kiedy metoda

`run` się skończy, wątek umiera, kończy swoją pracę. Ponieważ sygnatura metody `run` jest już zdefiniowana w klasach `Thread` oraz `Runnable` (i wygląda następująco: `public void run(){...}`). Do metody `run` nie można przekazać argumentów, nie można zwrócić z niej żadnej wartości - nie można zmienić jej sygnatury. Należy więc w inny sposób przekazać jej zmienne, na których będzie mogła pracować - najprościej zrobić to za pomocą przekazywania zmiennych w konstruktorze. Metoda `run` jest dla wątku tym, czym dla niewątkowego, tradycyjnego programu jest funkcja `main`. Tradycyjny, nie-wątkowy program kończy swoje działanie wraz z zakończeniem funkcji `main`. Program wielowątkowy kończy swoje działanie gdy jednocześnie funkcja `main` oraz metoda `run` kończą swoje działanie.

Instancja klasy `java.net.ServerSocket` występuje tylko po stronie serwera (reprezentuje gniazdo serwerowe TCP) i nasłuchuje na klientów i tworzy instancje klasy `java.net.Socket` (gniazdo klienckie TCP) do obsługi połączeń przychodzących od klientów. `java.net.ServerSocket` to klasa reprezentująca gniazdo oczekujące na przychodzące żądania połączeń. Gniazdo serwerowe ma za zadanie nasłuchiwania na podanym porcie zgłoszeń klientów i następnie obsługiwanie połączenia z nim.

Metoda `close()` gniazda (klasy zarówno `java.net.Socket` jak i `java.net.ServerSocket`) zamyka automatycznie jego strumienie do komunikacji. Gdy jeden ze strumieni do komunikacji zostanie zamknięty, całe gniazdo jest zamykane. Gdy w czasie pracy z gniazdem chcemy zamknąć tylko strumień do czytania lub pisania należy użyć jednej z metod: `shutdownInput()`, `shutdownOutput()`.

Uwaga W poniższych zadaniach zakładamy, iż serwer powinien obsługiwać kilku klientów w danej chwili.

1. Napisz program serwera, który działając pod adresem 127.0.0.1 oraz na określonym porcie TCP będzie serwerem echa, który będzie odsyłał podłączającym się klientom odebrane od nich wiadomości.

2. Zmodyfikuj program 1 z laboratorium 12 w taki sposób, aby serwer zapisywał do pliku wszystkie zdarzenia, wraz z czasem ich zajścia. Przykładowo: serwer ma zapisywać do pliku datę i czas połączenia nowego klienta, oraz jego adres i port, z którego dany klient łączy się z serwerem.

3. Napisz program serwera, który działając pod adresem 127.0.0.1 oraz na określonym porcie TCP będzie losował liczbę i odbierał od klienta wiadomości. W przypadku, gdy w wiadomości klient przyśle do serwera coś innego, niż liczbę, serwer powinien poinformować klienta o błędzie. Po odebraniu liczby od klienta, serwer sprawdza, czy otrzymana liczba jest:

   - mniejsza od wylosowanej przez serwer
   - równa wylosowanej przez serwer
   - większa od wylosowanej przez serwer

   A następnie odsyła stosowną informację do klienta. W przypadku, gdy klient odgadnie liczbę, klient powinien zakończyć działanie.

4. Zmodyfikuj swoją aplikację zaliczeniową w taki sposób, aby serwer mógł obsługiwać kilku klientów jednocześnie (przekształć swój serwer w serwer wielowątkowy).