Zadanie laboratoryjne nr 5

Programowanie Współbieżne
2014-12-07

Łukasz Ochmański 183566
Marcel Wieczorek 173526

Sprawozdanie

Do przedstawienia działania monitora napisałem aplikację typu klient-serwer, gdzie dwie grupy wątków programu w losowych odstępach czasu korzystają z określonego zasobu. Losowość jest zapewniana przez scheduler systemu operacyjnego, gdyż wątki są wybierane w niedeterministyczny sposób i zależą od nieprzewidywalnych czynników, tj. połączenie z internetem. Ruch sieciowy jest na tyle nieprzewidywalny i może wpływać na działanie maszyny tak mocno, że można uznać, że warunki w jakich scheduler wybiera z pośród tysiąca wątków są wystarczająco przypadkowe, aby wykonać to zadanie laboratoryjne. Pierwsza grupa wątków używa zasobu dziesięciokrotnie częściej i dłużej niż druga grupa wątków. Aby zasymulować dłuższe działanie jednej z grup wątków użyłem funkcji sleep(long time), która nie zawiesza działania wątku w taki sposób, jak robi to wait(). Stan wątku zostaje niezmieniony (wciąż runnable) i monitor nie zostaje oddany. Zostaje w posiadaniu wątku do końca trwania sleep(). Czekanie nie odbywa się wewnątrz monitora, więc wątki imitują procesowanie obliczeń równolegle, co zwiększa wydajność całego systemu.

```
public class Klient extends Thread {
    private Zasób zasób;
    private long pauza;
    private int znaków;

    Klient(Zasób b, long pauza, int znaków) {
        this.zasób = b;
        this.pauza = pauza;
        this.znaków = znaków;
        b.addTaskCount(znaków);
    }

    public void run() {
        for (int i = 0; i < znaków; i++) {
            zasób.put();
            try {
                sleep(pauza);
            } catch (InterruptedException e) {
                Logger.getGlobal().info("Ktoś przerwał mój sen!");
            }
        }
    }
}
```

Rolę monitora, który odpowiedzialny jest za sprawiedliwy dostęp do zasobów będzie pełniła napisana przeze mnie klasa Zasób.java, która również zawiera tablicę znaków, która będzie blokowała dostęp do zasobu.

```
public class Zasób {
    private char[] zasób;
    private int count, in, out, literka;
    private static int tasks;

    public Zasób(int size) {
        zasób = new char[size];
    }

    public synchronized void put() {
        while (count == zasób.length) {
            Logger.getGlobal().info("Zasób jest pełny, więc usypiam wątek.");
            try {
                wait();
            } catch (InterruptedException e) {
                Logger.getGlobal().info("Nastąpiło przerwanie w metodzie put");
            }
        }
        char c =(char) ('A' + literka % 26);
        Logger.getGlobal().info("-->> " + c + " -->>");
        zasób[in] = c;
        in = (in + 1) % zasób.length;
        literka++;
        count++;
        notifyAll();
    }

    public synchronized char take() {
        while (count == 0) {
            Logger.getGlobal().info("Zasób jest pusty, więc usypiam wątek.");
            try {
                wait();
            } catch (InterruptedException e) {
                Logger.getGlobal().info("Nastąpiło przerwanie w metodzie take");
            }
        }
        char c = zasób[out];
        out = (out + 1) % zasób.length;
        count--;
        tasks--;
        Logger.getGlobal().info("<<-- " + c + " <<--");
        notifyAll();
        return c;
    }

}
```

W zadaniu użyłem dwóch grup wątków reprezentujących klientów. Pierwsza grupa wątków używa zasobów dziesięciokrotnie częściej i dłużej. Każdy wątek działa 100ms i przesyła żądanie 100 razy.

```
for (int i=0; i<10; i++) {
    //duration = 100ms
    //calls = 100;
    wątki[i] = new Klient(z, 100, 100);
}
```

Druga grupa wątków używa zasobów dziesięciokrotnie rzadziej i krócej. Każdy wątek działa 10ms i przesyła żądanie tylko 10 razy.

```
for (int i=10; i<20; i++) {
    //duration = 10ms
    //calls = 10;
    wątki[i] = new Klient(z, 10, 10);
}
```

Sprawność obliczyłem ze wzoru poniżej. Do obliczeń założyłem, że mój program utworzy 20 wątków stanowiących klientów i jeden wątek będący serwerem. Jak łatwo zauważyć w sumie wątki zlecą wykonanie 1100 żądań (100*10+10*10). Przy zlecaniu jednego zadania każdy wątek wykona około 30 instrukcji assemblerowych, z czego około 60% to operacje wejścia/wyjścia, a 40% to operacje na rejestrach.

ω(n) = 40% z 33000 (1100*30) operacji obliczeniowych, ponieważ tyle jest operacji na rejestrach

h(n,p) = 60% z 33000 operacji we/wy ponieważ większość operacji to przepisywanie wartości z jednej komórki pamięci do drugiej. W moim programie ma tam szczególnie skomplikowanych obliczeń.

Zatem:

$$\eta(n,p) = \frac{\omega(n)}{\omega(n) + h(n,p)} = \frac{0{,}4 * 33000}{0{,}4 * 33000 + 0{,}6 * 33000} = \frac{4}{10} = 40{,}00\%$$

Wnioski:

40% jest to imponujący dobry wynik, ponieważ programy typu klient serwer charakteryzyją się dużą wydajnością. Gdyby uwzględnić czas czekania w metodzie sleep(), poza monitorem i policzyć to jako operacje obliczeniowe, wydajność systemu mogłaby wynieść nawet 90%. Jeśli weźmiemy pod uwagę fakt, że wątki klientów są wielokrotnie wolniejsze, a wątek serwera jest wielokrotnie szybszy, nagle zdajemy sobie sprawę, że sytuacja jest nieco inna niż w zadaniu laboratoryjnym nr 2. Tam wąskim gardłem była magistrala pamięci, która była jedna i wolna. Tutaj wąskie gardło czyli serwer jest szybki, a klienci są wolni. Serwer jest w stanie przetworzyć żądania szybciej niż są w stanie przychodzić. Serwer nie musi współdzielić zasobów z nikim innym. Jego zadania są kolejkowane i nieprzerwanie obsługiwane na najwyższych obrotach. Zaletą tego rozwiązania jest to, że nie musi dzielić z nikim zasobów jest autonomiczny. Im więcej zadań przychodzi do serwera, tym większa sprawność całego systemu. Zatem ta architektura będzie miała większą sprawność niż komputer osobisty z jedną wolną magistralą pamięci

i wieloma szybkimi procesorami. Gdyby jednak klienci byli szybsi od serwera, sytuacja byłaby równie beznadziejna co z zadaniu laboratoryjnym nr 2.

W istocie monitor jest obiektową wersja zamka, który działa per obiekt i jest przekazywany z wątku do wątku. Każdy wątek, który chce wywołać metodę, musi przejść przez monitor, aby wykonać swoje zadanie. W ten sposób monitor może kontrolować, komu, kiedy i jak, przydzielić zasoby. Główną zaletą monitora jest możliwość dodawania warunków, które mogą być wykorzystane do implementacji sprawiedliwego dostępu. Monitor może zawierać na przykład kolejkę FIFO czyli BlockingQueue lub PriorityQueue. Monitor przypomina swoim działaniem planistę (ang. Scheduler) systemu operacyjnego. Jest to jednak bardzo prosta struktura, która nie nasłuchuje na powiadomienia typu wait(), notify(), start(), stop(), itd. Scheduler ściśle współpracuje z programem, który może mieć w sobie zaawansowany monitor. Programista nie ma dostępu do schedulera, ale ma dostęp do monitora, który jest swojego rodzaju schedulerem w obrębie jednego processu.