Projekt

Systemy Operacyjne 2

środa TP, 11:15 - 13:00

Problem producenta i konsumera - Kasyno

prowadzący
Mgr inż. Mateusz Gniewkowski

wykonał:
Patryk Uzarowski, 259105

Informatyka Techniczna, W4N

19 kwietnia 2023

Spis treści

1 Wstęp

Wielowątkowość jest jednym z kluczowych pojęć z dziedziny programowania i informatyki. Odnosi się do możliwości jednoczesnego wykonywania wielu wątków programu, czyli jednostek przetwarzających, w obrębie jednego procesu. Dzięki temu program może działać szybciej i wydajniej, co ma szczególne znaczenie w przypadku aplikacji wymagających przetwarzania dużej ilości danych.

Jednym z problemów związanych z wielowątkowością jest problem producenta-konsumera, który polega na synchronizacji pracy dwóch lub więcej wątków, z których jeden produkuje dane, a drugi je konsumuje. W przypadku nieprawidłowej synchronizacji, może dojść do sytuacji, w której producent wyprodukuje więcej danych, niż jest w stanie je przetworzyć konsument, co prowadzi do nadmiernego zużycia zasobów systemowych i spadku wydajności całego programu.

W niniejszym sprawozdaniu dokładniej omówiony zostanie problem producenta-konsumenta na przykładzie głównego zadania projektowego - problemu synchronizacji kasyna, gdzie producenci - automatyczne podajniki kart, powinny adekwatnie współpracować z konsumentem - krupierem, który pobiera karty do dalszej gry.

2 Skrypt podstawowy

W ramach pierwszego etapu projektu został opracowany skrypt, który implementuje wstępne założenia projektu, takie jak strukturę kart, funkcje podajników oraz krupiera. Skrypt nie był jednak implementowany z myślą adekwatnej i wydajnej pracy na wielu wątkach, brakuje w nim wielu kluczowych elementów jak precyzyjna synchronizacja, kryteria stopu lub oczekiwań czy w końcu stosu wywołań oraz mutexów.

```
#include <iostream>
#include <queue>
#include <thread>
#include <random>

using namespace std;
using namespace this_thread;

const int NUMBER_OF_FEEDERS = 2;
const int NUMBER_OF_CARDS = 52;

struct Card {
    int suit; //Przyjeto: 0: pik, 1: kier, 2: karo, 3: trefl
    int rank; //Przyjeto: 1: as, 2-10: std, 11: jupek, 12: dama, 13: krol
    string cardToString(){
        string returnString = "card:";

        switch(rank){
            case 1:
                returnString += " Ace";
                break;
            case 11:
                returnString += " Jack";
                break;
            case 12:
                returnString += " Queen";
                break;
            case 13:
                returnString += " King";
                break;
            default: returnString += " " + to_string(rank);
        }

        switch(suit){
```

```
            case 0:
                returnString += " of spades";
                break;
            case 1:
                returnString += " of hearts";
                break;
            case 2:
                returnString += " of diamonds";
                break;
            case 3:
                returnString += " of clubs";
                break;
        }

        return returnString;
    }
};


queue<Card> buffer;
bool stop_threads = false;

void feeder(int id){
    printf("Starting feeder\n");

    random_device rd;
    mt19937 gen(rd());
    uniform_int_distribution<int> suit_dist(0, 3);
    uniform_int_distribution<int> rank_dist(1, 13);

    for(int i = 0; i <  NUMBER_OF_CARDS; i++){
        Card card;
        card.suit = suit_dist(gen);
        card.rank = rank_dist(gen);
        // cout<< "Card feeder " << id << " produces " << card.cardToString() << endl;
        printf("Card feeder %d, produces card: suit %d, rank %d\n", id, card.suit, card.
    rank);
        buffer.push(card);
    }

}

void dealer(){
    printf("Starting dealer\n");

    int count = 0; //liczba pobranych kart

    while(true){

        if(stop_threads && buffer.empty() && count == NUMBER_OF_CARDS) break;

        if(!buffer.empty()){
            Card card = buffer.front();
            buffer.pop();
            cout << "Dealer consumes " << card.cardToString() << endl;
            count++;
        }else{
            static int number_of_finished_feeders = 0;
            if(++number_of_finished_feeders == NUMBER_OF_FEEDERS){
                stop_threads = true;
                break;
            }
        }
    }
}

int main(){

    //Start
    printf("\n!Initialize casino!\n");

    // Tworzenie tabele watkow podajnik w kart na podstawie stalej skryptu
    // w etapie 1, uwzgledniany jest tylko 1 podajnik
```

```
107     thread feeders[NUMBER_OF_FEEDERS];
108     for(int i = 0; i< NUMBER_OF_FEEDERS; i++){
109         feeders[i] = thread(feeder,i);
110     }
111
112     thread dealer_thread(dealer);
113
114     for(int i = 0; i<NUMBER_OF_FEEDERS; i++){
115         feeders[i].join();
116     }
117     dealer_thread.join();
118
119     printf("\n!End casino!\n");
120     return 0;
121 }
```

Kluczowe elementy skryptu:

- struct Card - elementarna struktura projektu definiująca obiekt karty, które będą brały udział w funkcji podajników oraz krupiera. Struktura posiada dwa pola: **suit** - kolor karty oraz **rank** - wartość karty. W celu polepszenia prezentacji wyników zdecydowano się zaimplementować również metodę cardToString(), która odpowiada za adekwatne listowanie obiektów kart ze względu na ich odpowiadające wartości. Zdecydowano się przyjąć następujące założenia względem wartości logicznych pól:
  - Suit - 0: pik, 1: kier, 2: karo, 3: trefl
  - Rank - 1: as, 11: jupek, 12: dama, 13: król, 2-10: 2-10.
- feeder() - funkcja automatycznych podajników kart, odpowiada za pseudo losową generację kart z 52 elementowej talii. Przy każdej iteracji pętli głównej do kolejki programu **queue¡Card¿ buffer** dodawana jest kolejna wygenerowana karta, o jej charakterystyce możemy dowiedzieć się z poziomu konsoli - funkcja printf().
- dealer() - funkcja krupiera odpowiadająca za pobieranie kart ze szczytu kolejki oraz informowanie o charakterystyce karty oraz numerze podajnika, z którego karta została odebrana. Kryterium stopu pętli głównej jest określone poprzez:
  - Zmienna globalna stop_threads informująca o wymaganym zatrzymaniu wątku.
  - buffer.empty() - w przypadku pustej kolejki dopuścilibyśmy do jednego z podstawowych błędów problemu producenta-konsumenta - zagłodzenia wątku krupiera
  - count - zatrzymanie pracy wątku w przypadku pobrania wszystkich możliwych kart z podajników.
- main() - funkcja rozruchowa skryptu odpowiadająca za stworzenie wątków podajników, wątku krupiera oraz ich uruchomienie.

Powyższy skrypt nie radzi sobie w przypadku wyższej liczby podajników kart, jest to spowodowane brakiem implementacji mechanizmów kontrolujących przepływ informacji pomiędzy uruchomionymi wątkami. Brak adekwatnej synchronizacji może prowadzić do wielu problemów takich jak wyścigi, zagłodzenia czy overflow wspólnego źródła danych. W kolejnym etapie projektu skrypt zostanie przebudowany w celu wykluczenia powyższych zagrożeń oraz prawidłowej implementacji oprogramowania w celu przetestowania jego działania na dużej ilości wątków podajników kart.

3 Rozbudowa skryptu o mechanizmy zabezpieczające

Skrypt z poprzedniego punktu został adekwatnie rozbudowany o mechanizmy wielowątkowości usprawniające i optymalizujące pracę programu.

```
#include <iostream>
#include <queue>
#include <thread>
#include <random>
#include <mutex>
#include <condition_variable>

using namespace std;
using namespace this_thread;
using namespace std::chrono;

const int NUMBER_OF_FEEDERS = 4;
const int NUMBER_OF_CARDS = 52;

struct Card {
    int suit; //Przyjeto: 0: pik, 1: kier, 2: karo, 3: trefl
    int rank; //Przyjeto: 1: as, 2-10: std, 11: jupek, 12: dama, 13: krol
    string cardToString(){
        string returnString = "card:";

        switch(rank){
            case 1:
                returnString += " Ace";
                break;
            case 11:
                returnString += " Jack";
                break;
            case 12:
                returnString += " Queen";
                break;
            case 13:
                returnString += " King";
                break;
            default: returnString += " " + to_string(rank);
        }

        switch(suit){
            case 0:
                returnString += " of spades";
                break;
            case 1:
                returnString += " of hearts";
                break;
            case 2:
                returnString += " of diamonds";
                break;
            case 3:
                returnString += " of clubs";
                break;
        }

        return returnString;
    }
};


queue<Card> buffer;
// Wprowadzenie mutexu w celu poprawy synchronizacji, zapobieganie wyscigom
mutex buffer_mutex;
//Zmienna warunkowa poprawia synchronizacje, inicjalizuje warunek stopu
condition_variable buffer_cv;
int finished_feeders = 0;
//Bezwzgledny warunek stopu
bool stop_threads = false;

```

```
66  void feeder(int id){
67      printf("Starting feeder\n");
68
69      random_device rd;
70      mt19937 gen(rd());
71      uniform_int_distribution<int> suit_dist(0, 3);
72      uniform_int_distribution<int> rank_dist(1, 13);
73
74      for(int i = 0; i <  NUMBER_OF_CARDS; i++){
75          Card card;
76          card.suit = suit_dist(gen);
77          card.rank = rank_dist(gen);
78
79          unique_lock<mutex> lock(buffer_mutex);
80          buffer_cv.wait(lock, [] {return buffer.size() < NUMBER_OF_FEEDERS ||
        stop_threads; });
81
82          if(stop_threads) break;
83          cout<< "Card feeder " << id << " produces " << card.cardToString() << endl;
84          // printf("Card feeder %d, produces card: suit %d, rank %d\n", id, card.suit,
        card.rank);
85          buffer.push(card);
86          // Odblokowanie mutexu
87          lock.unlock();
88          // Powiadomienie watku konsumera
89          buffer_cv.notify_one();
90      }
91
92      {
93          // Mutex guarda nadzorujacy ewentualny warunek stopu
94          lock_guard<mutex> lock(buffer_mutex);
95          finished_feeders++;
96          if(finished_feeders == NUMBER_OF_FEEDERS){
97              stop_threads=true;
98              buffer_cv.notify_one();
99          }
100     }
101
102 }
103
104 void dealer(){
105     printf("Starting dealer\n");
106
107     int count = 0; //liczba pobranych kart
108
109     while(true){
110
111         unique_lock<mutex> lock(buffer_mutex);
112
113         buffer_cv.wait(lock, [] {return !buffer.empty() || (stop_threads &&
        finished_feeders == NUMBER_OF_FEEDERS); });
114
115         if(stop_threads && buffer.empty() && count == NUMBER_OF_CARDS *
        NUMBER_OF_FEEDERS) break;
116
117         if(!buffer.empty()){
118             Card card = buffer.front();
119             buffer.pop();
120             cout << "Dealer consumes " << card.cardToString() << endl;
121             count++;
122             lock.unlock();
123             buffer_cv.notify_one();
124         }else{
125             static int number_of_finished_feeders = 0;
126             if(++number_of_finished_feeders == NUMBER_OF_FEEDERS){
127                 stop_threads = true;
128                 lock.unlock();
129                 break;
130             }
131             lock.unlock();
132         }
133     }
134 }
```

```
135
136 int main(){
137
138     //Start
139     printf("\n!Initialize casino!\n");
140
141     // Tworzenie tabele watkow podajnikow kart na podstawie stalej skryptu
142     // w etapie 1, uwzgledniany jest tylko 1 podajnik
143     thread feeders[NUMBER_OF_FEEDERS];
144     for(int i = 0; i< NUMBER_OF_FEEDERS; i++){
145         feeders[i] = thread(feeder,i);
146     }
147
148     thread dealer_thread(dealer);
149
150     for(int i = 0; i<NUMBER_OF_FEEDERS; i++){
151         feeders[i].join();
152     }
153     dealer_thread.join();
154
155     buffer_cv.notify_all();
156
157     printf("\n!End casino!\n");
158     return 0;
159 }
```

Do kluczowych modernizacji oraz usprawnień zaimplementowanych mechanizmów należą:

- Mutex - Mechanizm mutex (buffer_mutex) jest używany do synchronizacji wątków, aby zapobiec sytuacjom wyścigu (race conditions) podczas dostępu do kolejki buffer. W tym przypadku mutex jest używany do blokowania dostępu do buffer przez wiele wątków jednocześnie.

- Condition variable - Funkcja używa zmiennej warunkowej (buffer_cv) do synchronizacji wątków i informowania wątków o zmianie stanu. Wątki zostają zawieszone w momencie, gdy buffer jest pełny (wielkość równa NUMBER_OF_FEEDERS) lub gdy zmienna stop_threads jest ustawiona na true. Wątki zostają ponownie uruchomione, gdy jeden z wątków pobierze obiekt Card z kolejki buffer, co powoduje zmniejszenie liczby elementów w kolejce.

- Zmienna stop_threads - Ta zmienna służy do kończenia pracy wątków. Jeśli zmienna stop_threads jest ustawiona na true, wątki kończą swoje działanie.

- Zmienna finished_feeders - Ta zmienna służy do śledzenia ilości wątków, które zakończyły swoją pracę. Po tym, jak ostatni wątek skończy pracę, zmienna stop_threads jest ustawiana na true i informuje to pozostałe wątki, że mają kończyć swoją pracę.

Dokładniejszy opis zaimplementowanego mechanizmu synchronizacji wątków:
Do synchronizacji dostępu do kolejki buffer wykorzystane zostały mechanizmy mutex oraz condition_variable. Mutex zapewnia wzajemne wykluczanie, czyli blokowanie dostępu do zasobu, gdy jest on aktualnie używany przez inny wątek. Natomiast condition_variable pozwala na powiadamianie wątków o zdarzeniach związanych z zasobem, do którego przypisana jest kolejka buffer. W tym przypadku, gdy liczba elementów w kolejce buffer osiągnie wartość mniejszą niż liczba podajników, wątki feeder zawieszą się na condition_variable i czekają na powiadomienie przez wątek dealer. Gdy liczba elementów w kolejce buffer osiągnie wartość równą liczbie podajników, wątek dealer pobierze element z kolejki i powiadomi o tym wątek feeder.

4 Wnioski

Opracowany skrypt problemu producenta-konsumenta działa poprawnie. Rozbudowa skryptu podstawowego o kluczowe mechanizmy zabezpieczeń działania programu pozwoliła na bezproblemowe uruchomienie programu kasyna na dużej liczbie wątków podajników kart. Zaprezentowany problem jest jednym z najczęściej spotykanych problemów synchronizacji w programowaniu wielowątkowym. Zastosowanie odpowiednich technik synchronizacji, takich jak mutex i warunkowa zmienna umożliwiło bezpieczne i efektywne rozwiązanie tego zadania. Warto zwrócić uwagę, że dobór liczby wątków jest istotnym czynnikiem wpływającym na wydajność programu i powinien być dokładnie opracowany.