Documentatie lab 5 PPD

Analiza cerintelor

La un concurs de programare participa mai multi concurenti din diferite tari. Concursul presupune rezolvarea mai multor(10) probleme intr-un interval de timp dat. Fiecare participant este identificat printr-un ID si rezultatele obtinute pentru rezolvarea fiecarei probleme sunt salvate intr-un fisier cu rezultate in care pe fiecare linie sunt inregistrari de forma (ID, Punctaj, Tara). Punctajele concurentilor dintr-o tara C1 corespunzatoare fiecarei probleme se gasesc in fisiere separate (“RezultateC1_P1.txt”, “RezultateC1_P2.txt, …“RezultateC1_P10.txt). Un punctaj negativ (-1) inseamna incercare de fraudare si va conduce la eliminarea concurentului din concurs. Daca un concurent nu rezolva o problema nu se adauga in fisier o pereche cu ID –ul concurentului si punctaj 0.

Pentru a se ajunge la clasamentul final se cere sa se formeze o lista inlantuita ordonata descrescator (dupa punctaj) care contine elemente cu valori de tip pereche (ID, Punctaj, Tara). Lista poate fi ordonata doar la final.

Se porneste prin crearea unei liste inlantuita vida si se adauga elemente sau punctaje pe masura ce se citesc inregistrari din fisiere.

La citirea unei noi perechi (ID_n, Punctaj_n) se verifica daca exista deja in lista o pereche cu ID egal cu ID_n:

- Daca exista atunci
  - Daca Punctaj_n este pozitiv se aduna punctajul Punctaj_n la punctajul existent in nod,
  - Daca Punctaj_n = -1 atunci se sterge nodul gasit din lista
    - Daca se va citi ulterior o pereche cu acelasi ID nu se va mai adauga in lista
- Daca nu exista atunci se adauga un nou nod cu valoarea (ID_n, punctaj_n).

In final aceasta lista va conține clasamentul final si se va salva intr-un fisier „Clasament.txt” de catre main-thread.

Implementare paralela – p threaduri

1. **p_r** threaduri (readers) citesc din fisiere perechi (ID, Punctaj) si le adauga intr-o structura de date de tip coada, marginita, cu capacitate maxima 50 sau 100.
   (conditie - pentru structura de tip coada NU se admite folosirea unei structuri de date pentru care partea de sincronizare este deja implementata)
2. Celelalte threaduri **p_w** = (p - p_r) threaduri (workers) preiau cate o pereche din coada si o adauga in lista L; Se continua operatiile 1., 2. pana cand toate perechile, din toate fisierele, sunt adaugate la lista L.
3. Dupa ce toate rezultatele au fost adaugate in lista L, primul thread (primul reader) scrie rezultatul obtinut in lista L in fisierul „Clasament.txt”.

Atentie: O stare in care coada(queue) vida nu inseamna neaparat ca toate inregistrarile au fost citite si introduse in lista!

Se doreste folosirea variabilelor conditionale pentru implementarea “producator-consumator”.
Obiectiv:

1. Intelegerea/aprofundarea sablonului “producator-consumator”
2. Intelegerea/aprofundarea sincronizarii conditionale
3. Intelegerea/aprofundarea excluderii mutuale (granularitatea sectiunilor critice)
4. Fine-grained synchronization vs Coarse-grained synchronization

Proiectare

Ca si tip abstract de date vom folosi o lista simplu inlantuita, thread-safe, cu sentinele, neordonata. Lista va respecta fine-grained synchronization adica vom bloca cat mai putin necesar pentru a face o operatie. Pentru procesarea intrarilor vom folosi un tip abstract de date de tip coada thread-safe cu capacitate maxima, iar sincronizarea o vom face cu ajutorul variabilelor conditionale. Numarul de threaduri (configuratia) se va transmite prin intermediul parametrilor.

Detalii de implementare

```
VOID
ProducerWorker(
    _Inout_ ProducerConsumerQueue<Participant>& PcQueue,
    _In_ const List<Participant>&               Participants,
    _In_ int ThreadIndex = 0
)
{
    for (int i = 1; i <= COUNTRY_COUNT; ++i)
    {
        for (int j = 1 + ThreadIndex; j <= TASKS_COUNT; j += gProducersCount)
        {
            CHAR filePath[MAX_PATH] = { 0 };
            _snprintf_s(filePath, MAX_PATH, _TRUNCATE, "D:\\facultate-repo\\sem5\\ppd\\lab 5\\inputs\\Rez

            std::ifstream fin(filePath);
            if (!fin.is_open())
            {
                continue;
            }

            int participantId, participantScore;
            while (fin >> participantId >> participantScore)
            {
                Participant participant;
                participant.SetId(participantId);
                participant.SetScore(participantScore);
                participant.SetCountry(i);
                PcQueue.Produce(participant);
            }
            fin.close();
        }
    }

    PcQueue.UnregisterProducer();

    if (ThreadIndex == 0)
    {
        while (PcQueue.IsAnyConsumerActive())
        {
            std::this_thread::sleep_for(std::chrono::nanoseconds(1));
        }

        PrintResults(Participants);
    }
}
```

`ProducerWorker` este functia pe care o folosesc workerii (thread-urile) ce citesc din fisier. `PcQueue.Produce(participant);` asteapta sa poata bloca lacatul inainte de a actualiza coada, astfel mai multi workeri pot lucra in siguranta pe aceeasi coada in acelasi timp. Mai mult decat atat, workerii nu citesc de 2 ori aceleasi intrari din fisiere datorita `for (int j = 1 + ThreadIndex; j <= TASKS_COUNT; j += gProducersCount)` care asigura faptul ca fiecare worker citeste fisiere diferite. La final, dupa ce fiecare worker isi termina treaba, acesta anunta acest lucru prin `PcQueue.UnregisterProducer();`, iar prin functia `bool IsAnyProducerActive();` ne asteptam sa ni se returneze daca toti producerii au terminat de citit si inserat intrari din fisiere. La final, thread-ul 0 asteapta ca toti consumerii sa isi termine ce au de facut prin apelul la `bool IsAnyConsumerActive();` si apoi se scriu rezultatele in fisier prin apelul la `PrintResults(Participants);`.

```
VOID
ConsumerWorker(
    _Inout_ ProducerConsumerQueue<Participant>& PcQueue,
    _Inout_ List<Participant>&                  Participants
```

```
)
{
    while (true)
    {
        std::optional<Participant> participant = PcQueue.Consume();
        if (!participant.has_value())
        {
            break;
        }

        Participants.Process(participant.value());
    }

    PcQueue.UnregisterConsumer();
}
```

`ConsumerWorker` este functia pe care o folosesc workerii (thread-urile) ce proceseaza informatia citita din fisier. Acestia incearca in continuu sa “consume” prin apelul la `PcQueue.Consume();`, apel care returneaza obiectul daca a existat unul in coada, altfel returneaza `std::nullopt` daca coada este vida si nu mai este niciun producator activ. Procesarea obiectului se face la apelul `Participants.Process(participant.value());`. Functia din urma va bloca cate 2 noduri in acelasi timp, adica cate 2 participanti, mai exact participantul curent si urmatorul; acest lucru ne asigura ca functia este thread-safe si in acelasi timp blocam cat mai putin necesar. Este important de mentionat ca atunci cand este necesara o noua inserare, ultimul element trebuie si ramane blocat pana cand inserarea are loc; in acest fel un alt thread nu va putea bloca ultimul element pana cand acesta nu este actualizat de thread-ul care insereaza.
Cand un consumer thread se opreste, acest lucru va fi semnalat prin apelul la `PcQueue.UnregisterConsumer();`.

Testare si analiza performantei

C++

Numar produceri (p_r)	Tip calcul	Numar threaduri (p) (produceri + consumeri)	Timp executie (secunde)
4	Paralel	6	0.11700
		8	0.10874
		16	0.10799

Concluzii

Putem observa ca timpul s-a micsorat foarte mult fata de laboratorul precedent unde nu am folosit variabile conditionale si fine-grained synchronization. Din experimentele efectuate, putem observa ca numarul de consumatori nu mai are aproape deloc un impact, iar timpul de executie este aproape acelasi. Abordarea folosita pare ca este foarte eficienta in contextul problemei noastre.