Documentatie lab 4 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). 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). Lista este o lista ordonata (invariant: dupa orice operatie lista este ordonata).

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:

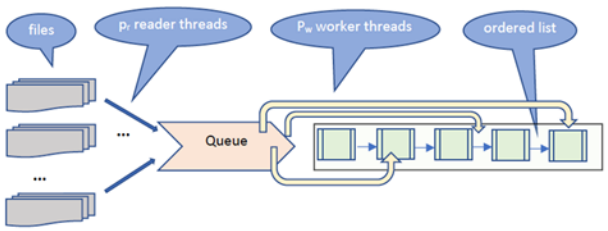
- Daca exista atunci
  - Daca Punctaj_n este pozitiv se aduna punctajul Punctaj_n la punctajul existent in nod,
  - Daca Punctaj_n = -1 atunci se sterge nodul gasit din lista
    - Daca se va citi ulterior o pereche cu acelasi ID nu se va mai adauga in lista
- Daca nu exista atunci se adauga un nou nod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

Metoda A) Implementare secventiala
Se citeste pe rand din fiecare fisier cate o pereche si se adauga in lista rezultat -L. La final se scriu elementele listei in fisierul rezultat.

Metoda B) 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 care nu este marginita (nu are capacitate maxim).
   (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 sa se foloseasca semafor/busy-waiting (chiar daca busy-waiting nu este o varianta foarte eficienta si prin urmare nici recomandata - intentia este de a face comparatie cu varianta cu variabile conditionale folosita la urmatorul laborator).
Obiectiv:

1. Intelegerea/aprofundarea sablonului “producator-consumator”
2. Intelegerea/aprofundarea sincronizarii conditionale
3. Intelegerea/aprofundarea excluderii mutuale (granularitatea sectiunilor critice)

Proiectare

Ca si structuri de date vom folosi un tablou unidimensional thread-safe care va ramane ordonat descrescator dupa fiecare actualizare si un tip abstract de date de tip coada thread-safe fara capacitate maxima. Numarul de threaduri (configuratia) se va transmite prin intermediul parametrilor.

Detalii de implementare

```
VOID
ProducerWorker(
    _Inout_ ProducerConsumerQueue<Participant>& PcQueue,
    _In_ const OrderedList<Participant>&        Participants,
    _In_ int ThreadIndex = 0
)
{
    for (int i = 1; i <= COUNTRY_COUNT; ++i)
    {
        for (int j = 1 + (ThreadIndex % (gProducersCount + 1)); j <= TASKS_COUNT; j += gProducersCount)
        {
            CHAR filePath[MAX_PATH] = { 0 };
            _snprintf_s(filePath, MAX_PATH, _TRUNCATE, "PATH\\inputs\\RezultateC%d_P%d.txt", i, j);

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

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

    PcQueue.UnregisterProducer();

    if (ThreadIndex == 0 && !gIsSecquential)
    {
        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 % (gProducersCount + 1)); j <= TASKS_COUNT; j += gProducersCount)` care asigura faptul ca fiecare worket 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. In cazul B, cand avem metoda folosind mai multe threaduri, 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_ OrderedList<Participant>&           Participants
)
{
    while (true)
    {
        if (!PcQueue.IsAnyProducerActive() && PcQueue.IsAllConsumed())
        {
            break;
        }

        std::optional<Participant> participant = PcQueue.Consume();
        if (!participant.has_value())
        {
            continue;
        }

        {
            std::lock_guard<std::mutex> lockGuard(gMutex);

            std::optional<Participant> existingParticipant = Participants.Get(participant.value());
            if (!existingParticipant.has_value())
            {
                if (participant.value().GetScore() == -1)
                {
                    participant.value().SetBlacklisted();
                }

                Participants.Insert(participant.value());
            }
            else
            {
                if (existingParticipant.value().IsBlacklisted())
                {
                    continue;
                }

                if (participant.value().GetScore() == -1)
                {
                    participant.value().SetBlacklisted();
                    Participants.Update(participant.value());
                }
                else if (participant.value().GetScore() > 0)
                {
                    participant.value().SetScore(participant.value().GetScore() + existingParticipant.val
                    Participants.Update(participant.value());
                }
            }
        }

        std::this_thread::sleep_for(std::chrono::nanoseconds(1)); // let other threads consume too
    }

```

```
56      PcQueue.UnregisterConsumer();
57  }
```

`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 dupa o nanosecunda nu a fost gasit niciun obiect in coada. In caz afirmativ, se incearca blocarea lacatului ce garanteaza faptul ca daca se proceseaza exact aceeasi 2 concurenti in acelasi timp, pentru probleme diferite, se obtin rezultatele dorite.
`std::this_thread::sleep_for(std::chrono::nanoseconds(1));` nu este foarte necesar, dar ajuta la “non-starvation”.
Un consumer thread se va opri atunci cand nu mai este niciun producator activ si cand nu mai este nimic de consumat, si 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)
1	Secvential	2	2.43011
1	Paralel	4	0.93927
		6	0.70905
		8	0.63844
		16	0.56390
2	Paralel	4	1.32357
		6	0.74602
		8	0.65365
		16	0.55589

Concluzii

Putem observa ca timpul s-a micsorat semnificativ atunci cand am folosit mai multe threaduri (in paralel) decat facand operatiile secvential. Chiar daca toate aceste obiecte folosite in procesarea paralela au un overhead, in toate cazurile abordarea paralela a fost mult mai rapida.
Putem face o comparatie chiar si intre abordarile paralele, in functie de numarul de threaduri folosite in total, dar si in functie de numarul de produceri; putem observa ca timpul de executie a scazut liniar atunci cand mai multe threaduri au fost folosite, iar cei mai buni timpi au fost obtinuti cand au fost folosite 16 threaduri in total. Numarul mai mare de produceri pare ca a afectat destul de putin timpii de executie, fiind cu un thread mai mult la producere si unul in minus la consumare.