

Міністерство освіти і науки України
Національний університет «Львівська політехніка»
Інститут комп'ютерних наук та інформаційних технологій
Кафедра Систем Штучного Інтелекту



Звіт
до лабораторної роботи № 10
з дисципліни
Операційні системи
на тему:
“ Синхронізація потоків в ОС Linux”

Виконав: студент КН-217

Ратушняк Денис

Прийняв: доцент каф. СШІ

Кривенчук Ю. П.

Мета роботи: Ознайомитися з особливостями синхронізації потоків в ОС Linux. Навчитися організовувати багатопоточність з використанням синхронізації в ОС Linux.

Завдання

1. [Макс. Складність 2] Модифікувати код лабораторної роботи №4 для виконання під ОС Linux.
2. [Складність відповідно до лаб. №4] Модифікувати код лабораторної роботи №4, щоб він став крос-платформенним - міг бути скомпільованим як під ОС Windows, так і під ОС Linux.

Код програми

```
#include <bits/stdc++.h>

#include <pthread.h>
#include <semaphore.h>
using namespace std;
typedef long long ll;
typedef long double ld;
typedef vector< vector<ll> > matrix;

///HERE YOU CAN CHANGE INPUT
const int TOTAL_THREADS = 100;
const int MAX_SEM_COUNT = 10;
string version = "10000_10000_10000";

ll type_of_task = 2;
ll low_priority_thread = 3;
ll high_priority_thread = 7;

///HERE YOU CAN CHANGE INPUT

typedef struct MyData
{
    int startx, starty, step, type;
    int threadNum;
} MYDATA, *PMYDATA;

MYDATA pDataArray[TOTAL_THREADS];

pthread_t dwThreadIdArray[TOTAL_THREADS];

matrix A;
ll n,m,k,max_number;
ll add;
vector < pair<ll,ll> > diag[TOTAL_THREADS];

vector< pair< pair<ll,ll>, pair<ll,ll> > > points_for_diag;

ld threadTime[TOTAL_THREADS];
pthread_mutex_t mymutex;
sem_t mysemaphore;
ll done_threads;
ll needThreads;

void solve(int sx, int sy, int step, int type, int num)
{
    if(type == 1)
```

```

{
    int i = sx;
    int j = sy;
    while(step--){
        diag[num][i+j].first += A[i][j];
        diag[num][i+j].second ++;
        diag[num][i-j+add].first += A[i][j];
        diag[num][i-j+add].second ++;
        //cout << i << " " << j << endl;
        //cout << num << " " << step << endl;
        j++;
        if(j == m){
            j = 0;
            i ++;
            if(i == n) break;
        }
    }
}
else
{
    int now = sx * n + sy;

    while(now < n * m){
        int i = now / m;
        int j = now % m;

        diag[num][i+j].first += A[i][j];
        diag[num][i+j].second ++;

        diag[num][i-j+add].first += A[i][j];
        diag[num][i-j+add].second ++;
        //cout << num << " " << now << " " << step << endl;
        now += step;
    }
}
ll ans = 0;
for(int i = 1 ; i < diag[num].size(); ++i)
    if(diag[num][i].first * diag[num][ans].second > diag[num][i].second * diag[num][ans].first ||
diag[num][ans].second == 0) ans = i;

ld result = (ld)diag[num][ans].first / (ld)diag[num][ans].second;
pthread_mutex_lock(&mymutex);
cout << "\033[F";
cout << "\033[F";
cout << "\n";
cout << "Tread " << num << " MaxAverage = " << result << "\n";
pthread_mutex_unlock(&mymutex);
}

void* MyThreadFunction(void *lpParam){
    sem_wait(&mysemaphore);
    PMYDATA pDataVar;

    pDataVar = (PMYDATA)lpParam;
    MYDATA DataVar = *pDataVar;

    std::chrono::time_point<std::chrono::system_clock> start, end;
    start = std::chrono::system_clock::now();

    solve(DataVar.startx, DataVar.starty, DataVar.step, DataVar.type, DataVar.threadNum);
}

```

```

end = std::chrono::system_clock::now();
std::chrono::duration<double> elapsed_seconds = end - start;
ld curtime = elapsed_seconds.count();

threadTime[pDataVar->threadNum] = curtime;

pthread_mutex_lock(&mymutex);

done_threads++;
cout << "\033[F";
cout << "\033[F";
cout << fixed << setprecision(3) << 100.0 * double(done_threads) / double(needThreads) << "%\n";
pthread_mutex_unlock(&mymutex);

sem_post(&mysemaphore);
return NULL;
}

void print(matrix &a)
{
    for(int i = 0; i < a.size(); ++ i)
    {
        for(int j = 0; j < a[i].size(); ++ j)
        {
            cout << a[i][j] << " ";
        }
        cout << "\n";
    }
    cout << "\n";
}

void print(matrix &a, ofstream &output)
{
    for(int i = 0; i < a.size(); ++ i)
    {
        for(int j = 0; j < a[i].size(); ++ j)
        {
            output << a[i][j] << " ";
        }
        output << "\n";
    }
    output << "\n";
}

void read(matrix &a, ifstream &input)
{
    for(int i = 0; i < a.size(); ++ i)
    {
        for(int j = 0; j < a[i].size(); ++ j)
        {
            input >> a[i][j];
        }
    }
}

void calc_points()
{
    points_for_diag.resize(2 * m + 2 * n - 2);
    ll x1,y1,x2,y2;
    x1 = y1 = x2 = y2 = 0;
}

```

```

for(int i = 0; i < (2 * m + 2 * n - 2)/2; ++i)
{
    points_for_diag[i] = {{x1, y1}, {x2, y2}};
    if(x1 == n - 1) y1++;
    else x1++;

    if(y2 == m - 1) x2++;
    else y2++;
}

x1 = x2 = 0;
y1 = y2 = m-1;

for(int i = (2 * m + 2 * n - 2)/2; i < (2 * m + 2 * n - 2); ++i)
{
    points_for_diag[i] = {{x1, y1}, {x2, y2}};
    if(y1 == 0) x1++;
    else y1--;

    if(x2 == n - 1) y2--;
    else x2++;
}
}

int main()
{
    #ifdef _WIN32
    string test_path = "A:\\T\\3_term\\Operating_Systems\\OSLabs\\10lab\\tests\\" + version + "_in.txt";
    #endif

    #ifdef __linux__
    string test_path = "/home/denisr2007/QT/OSLabs/10lab/tests/" + version + "_in.txt";
    #endif

    pthread_mutex_init(&mymutex, NULL);
    sem_init(&mysemaphore, 0, MAX_SEM_COUNT);
    low_priority_thread %= TOTAL_THREADS;
    high_priority_thread %= TOTAL_THREADS;
    if(low_priority_thread == high_priority_thread) low_priority_thread--;
    if(low_priority_thread == -1) low_priority_thread = 1;
    if(low_priority_thread == TOTAL_THREADS) low_priority_thread = 0;

    ifstream input(test_path);
    ld t0 = clock();
    input >> n >> m >> max_number;
    calc_points();
    add = n + 2*m - 2;

    A.resize(n);
    for(int i = 0; i < n; ++i) A[i].resize(m,0);

    read(A, input);
    //print(A);

    ll step1 = (n * m / TOTAL_THREADS);
    if(!step1) step1++;

    std::chrono::time_point<std::chrono::system_clock> start, end;
    start = std::chrono::system_clock::now();

    ld t1 = clock();

```

```

ll now1 = 0;
ll now2 = 0;

needThreads = min(ll(n * m), ll(TOTAL_THREADS));
ll step2 = needThreads;

for(int i = 0; i < needThreads; ++i)
{
    diag[i].resize(2 * n + 2 * m - 2, {0, 0});

    pdataArray[i].threadNum = i;
    pdataArray[i].type = type_of_task;

    if(type_of_task == 1)
    {
        pdataArray[i].startx = now1 / m;
        pdataArray[i].starty = now1 % m;
        pdataArray[i].step = step1;
        now1 += step1;
    }
    else
    {
        pdataArray[i].startx = now2 / m;
        pdataArray[i].starty = now2 % m;
        pdataArray[i].step = step2;
        now2 ++;
    }

    pthread_create(&dwThreadIdArray[i], NULL, &MyThreadFunction, &pdataArray[i]);
}
for(int i = 0; i < needThreads; ++i) pthread_join(dwThreadIdArray[i], NULL);
cout << "\n";
pthread_mutex_destroy(&mymutex);
sem_destroy(&mysemaphore);

for(int i = 1; i < needThreads; ++i)
{
    for(int j = 0; j < diag[0].size(); ++j){
        diag[0][j].first += diag[i][j].first;
        diag[0][j].second += diag[i][j].second;
    }
}

ll ans = 0;
for(int i = 1 ; i < diag[0].size(); ++i)
    if(diag[0][i].first * diag[0][ans].second > diag[0][i].second * diag[0][ans].first) ans = i;

end = std::chrono::system_clock::now();
std::chrono::duration<double> elapsed_seconds = end - start;
ld dectime = elapsed_seconds.count();

ld result = (ld)diag[0][ans].first / (ld)diag[0][ans].second;

#ifdef _WIN32
    string result_path = "A:\\T\\3_term\\Operating_Systems\\OSLabs\\10lab\\results_prob_right\\" +
to_string(TOTAL_THREADS);
#endif // _WIN32

#ifdef __linux__

```

```

    string result_path = "/home/denisr2007/QT/OSLabs/10lab/results_prob_right/" +
to_string(TOTAL_THREADS);
    #endif // __linux__

    result_path += "threads_" + to_string(type_of_task) + "typeOfTask_" + to_string(dectime) + "s_" + version
+ "_out.txt";
    ofstream output(result_path);
    output << "Number of diagonal where average element is the biggest = " << ans << " \n";
    output << fixed << setprecision(5) << "Average number = " << result << "\n";
    output << "2 points of this diagonal\n";
    output << points_for_diag[ans].first.first+1 << " " << points_for_diag[ans].first.second+1 << " " <<
points_for_diag[ans].second.first+1 << " " << points_for_diag[ans].second.second+1 << "\n";

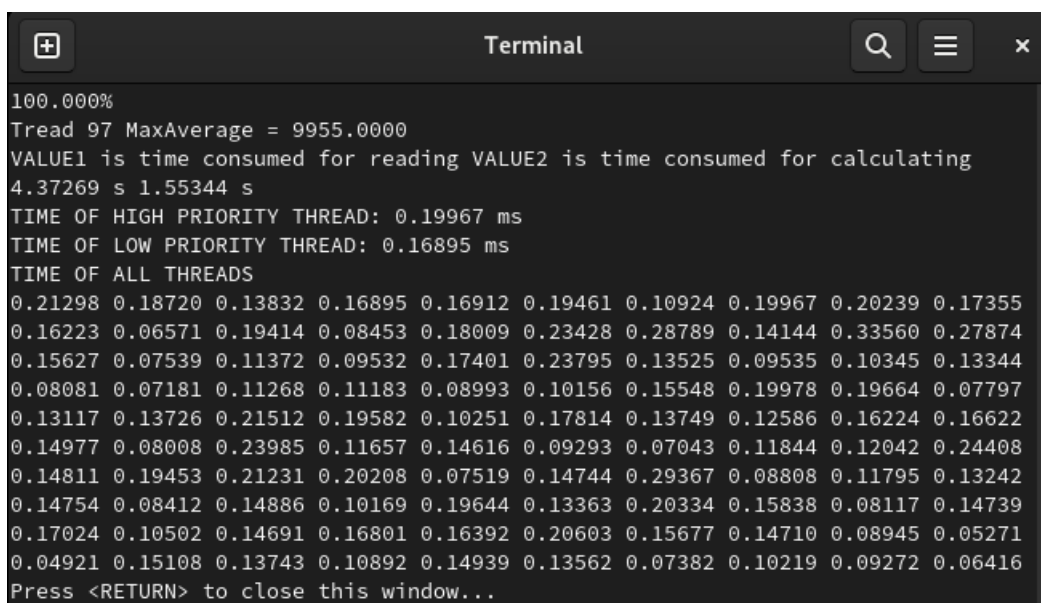
    ld d1 = t1 - t0;

    cout << "VALUE1 is time consumed for reading VALUE2 is time consumed for calculating" << endl;
    cout << fixed << setprecision(5) << d1/CLOCKS_PER_SEC << " s " << dectime << " s" << endl;
    cout << "TIME OF HIGH PRIORITY THREAD: " << threadTime[high_priority_thread] << " ms" << endl;
    cout << "TIME OF LOW PRIORITY THREAD: " << threadTime[low_priority_thread] << " ms" << endl;
    cout << "TIME OF ALL THREADS" << endl;
    for(int i = 0; i < TOTAL_THREADS; ++i) cout << threadTime[i] << " ";

}

```

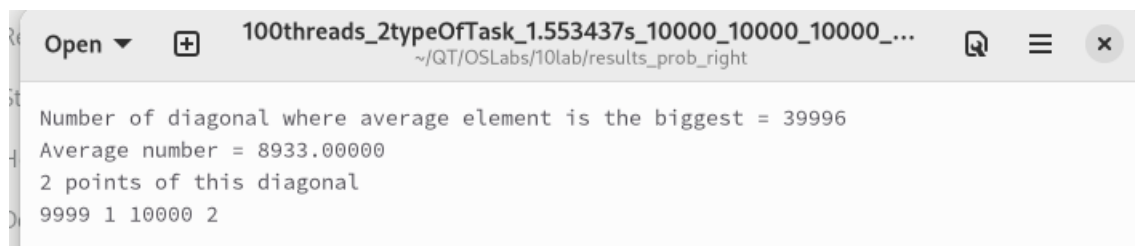
Результати



```

100.000%
Tread 97 MaxAverage = 9955.0000
VALUE1 is time consumed for reading VALUE2 is time consumed for calculating
4.37269 s 1.55344 s
TIME OF HIGH PRIORITY THREAD: 0.19967 ms
TIME OF LOW PRIORITY THREAD: 0.16895 ms
TIME OF ALL THREADS
0.21298 0.18720 0.13832 0.16895 0.16912 0.19461 0.10924 0.19967 0.20239 0.17355
0.16223 0.06571 0.19414 0.08453 0.18009 0.23428 0.28789 0.14144 0.33560 0.27874
0.15627 0.07539 0.11372 0.09532 0.17401 0.23795 0.13525 0.09535 0.10345 0.13344
0.08081 0.07181 0.11268 0.11183 0.08993 0.10156 0.15548 0.19978 0.19664 0.07797
0.13117 0.13726 0.21512 0.19582 0.10251 0.17814 0.13749 0.12586 0.16224 0.16622
0.14977 0.08008 0.23985 0.11657 0.14616 0.09293 0.07043 0.11844 0.12042 0.24408
0.14811 0.19453 0.21231 0.20208 0.07519 0.14744 0.29367 0.08808 0.11795 0.13242
0.14754 0.08412 0.14886 0.10169 0.19644 0.13363 0.20334 0.15838 0.08117 0.14739
0.17024 0.10502 0.14691 0.16801 0.16392 0.20603 0.15677 0.14710 0.08945 0.05271
0.04921 0.15108 0.13743 0.10892 0.14939 0.13562 0.07382 0.10219 0.09272 0.06416
Press <RETURN> to close this window...

```



```

Number of diagonal where average element is the biggest = 39996
Average number = 8933.00000
2 points of this diagonal
9999 1 10000 2

```

Висновок: Я закріпив вміння та навички роботи з синхронізацією потоків в ОС Linux. Навчився організовувати багатопоточність з використанням синхронізації в ОС Linux.