Laboratorium Podstawy Przetwarzania Rozproszonego

SPRAWOZDANIE z zadania 0

Nazwisko Imię	album	termin zajęć
Madajewski Adrian	145406	wtorek 16:50

Część I – Algorytm rozwiązania

1. Definicja problemu

Wyciąg narciarski o całkowitej nośności N kg wwozi na szczyt pojedynczych narciarzy. Każdy z S narciarzy waży łącznie ze sprzętem n_i kg (n_i < N lecz suma(n_i) > N). W każdym momencie suma wag n_i aktualnie zabranych narciarzy nie może przekroczyć N. Po dotarciu na szczyt narciarze przez pewien czas zjeżdżają na nartach, a następnie ponownie ubiegają się o wwiezienie na szczyt. Napisać program dla procesu narciarza, umożliwiający każdemu narciarzowi wielokrotne korzystanie z wyciągu.

2. Założenia przyjętego modelu komunikacji

- asynchroniczny system z wymianą komunikatów
- topologia połączeń: każdy z każdym
- wymagana pojemność kanału: S wiadomości w jednym kierunku
- inne wymagane własności sieci komunikacyjnej: kanały typu FIFO, transmisja rozgłoszeniowa

3. Algorytm wzajemnego wykluczania (event-driven)

W sekcji lokalnej:

- Narciarz zjeżdża losową ilość czasu ze stoku.

Aby wejść do sekcji krytycznej:

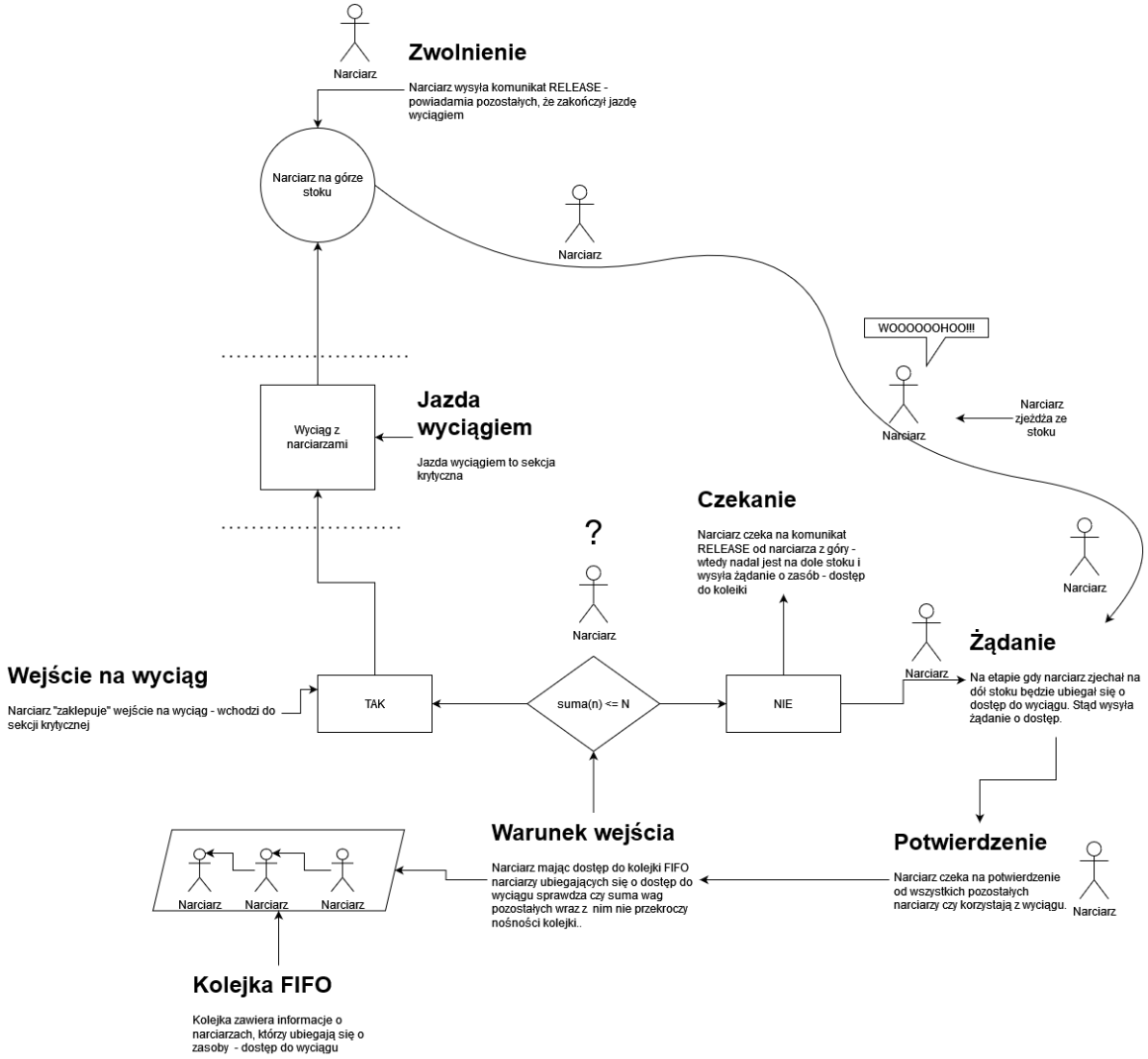
- Kiedy narciarz chce wjechać wyciągiem (sekcja krytyczna) musi wysłać żądanie (**REQUEST**) o strukturze: (id_procesu, stan_zegara, waga_narciarza) do każdego innego narciarza i wstawić je na swoje lokalną kolejkę.
- Kiedy narciarz otrzyma żądanie (**REQUEST**) od innego narciarza to odsyła etykietowaną czasem zegara odpowiedź (**REPLY**) z nowym czasem odpowiedzi i wstawia to otrzymane żądanie na swojej kolejce ubiegania się o zasób.

Wejście do sekcji krytycznej:

- Narciarz wchodzi na wyciąg (sekcja krytyczna) **TYLKO** jeśli etykietowana wiadomość **REPLY** otrzymana od każdego innego narciarza ma większą wartość zegara i jego żądanie o zasób do wyciągu jest pierwsze na jego lokalnej kolejce oraz spełniony jest warunek nie zajętości wyciągu (waga narciarzy musi być mniejsza niż nośność wyciągu) w przeciwnym wypadku narciarz czeka.

Wyjście z sekcji krytycznej:

- Kiedy narciarz opuszcza wyciąg (sekcje krytyczną) to usuwa swoje żądanie z kolejki żądań i wysyła do pozostałych narciarzy etykietowany stanem zegara komunikat **RELEASE** z informacją o zwolnieniu zasobu.
- Kiedy narciarz otrzymuje komunikat **RELEASE** od innego narciarza to usuwa go ze swojej lokalnej kolejki żądań do wyciągu.



4. Analiza złożoności komunikacyjnej algorytmu

złożoność pojedynczego przebiegu jednej instancji algorytmu (czyli z punktu widzenia pojedynczego procesu)

- Optymistyczna złożoność komunikacyjna pakietowa, wyrażona w liczbie komunikatów: 3(S-1)
- Optymistyczna złożoność czasowa przy założeniu jednostkowego czasu przesłania pojedynczego komunikatu w kanale: 3
- Pesymistyczna złożoność komunikacyjna pakietowa, wyrażona w liczbie komunikatów: 4(S-1)
- Pesymistyczna złożoność czasowa przy założeniu jednostkowego czasu przesłania pojedynczego komunikatu w kanale: 4

Część II – Implementacja rozwiązania

kod źródłowy implementacji w środowisku PVM/MPI

FILE.H

```
#ifndef FILE_H
#define FILE_H

#include <vector>
#include <string>
#include <fstream>
#include <iostream>

std::vector<int> loadDataFromFile(const std::string& filename);

#endif
```

FILE.CPP

```
#include "File.h"

std::vector<int> loadDataFromFile(const std::string& filename)
{
    std::vector<int> data;
    std::ifstream file;
    file.open(filename);

    if(!file.is_open())
    {
        std::cerr << "Couldn't find the file. Please restart." << '\n';
        exit(EXIT_FAILURE);
    }
    while(!file.eof())
    {
        int read;
        file >> read;
        data.emplace_back(read);
    }
    file.close();
    return data;
}
```

SKYER.H

```
#ifndef SKYER_H
#define SKYER_H

#include <vector>
#include <algorithm>
#include <mutex>
#include <memory>
#include <unistd.h> // for sleep()
#include <mpi.h>
#include <iomanip>
#include <thread>
```

```
extern int MAX_SKYERS;
extern int MAX_SKI_LIFT;
extern int MAX_SLEEP_TIME;
extern int MIN_SLEEP_TIME;

enum MPITag
{
    REQUEST = 0,
    REPLY,
    RELEASE,
};

struct Data
{
    int ID;
    int clock;
    int weight;
};

class Skyer
{
public:
    Data m_data;
    Skyer();
    void mainActivity();

private:
    // 2 threads per skier
    std::thread requestThread;
    std::thread releaseThread;

    // Mutexes
    std::mutex queueMutex;
    std::mutex clockMutex;
    std::mutex releaseMutex;

    // Resource queue
    std::vector<Data> queue;

    // Functionality
    void sortQueue();
    void addQueue(const Data &data);
    void deleteQueue(int id);
    int countAvailableWeight();
    void waitForSkiLift();
    void updateClock();
    void checkClock(const Data& recv_data);
    void localSleep();
    void requestForSkiLift();
    void waitForConfirm();
    void criticalSection();
    void releaseSkiLift();
};

#endif
```

SKYER.CPP

```
#include "Skyer.h"

#include <algorithm>

int MAX_SKYERS;
int MAX_SKI_LIFT;
int MAX_SLEEP_TIME;
int MIN_SLEEP_TIME;

// Set threads
Skyer::Skyer()
{
    releaseThread = std::thread([&](){
            int recv_ID;
            MPI_Status status;

            while (true)
            {
                MPI_Recv(&recv_ID, 1, MPI_INT, MPI_ANY_SOURCE, RELEASE, MPI_COMM_WORLD, &status);
```

```
                // Atomic
                updateClock();
                deleteQueue(recv_ID);

                releaseMutex.unlock();
            }
    });

    requestThread = std::thread([&](){
            int success = 1;
            int data[3];
            MPI_Status status;

            while(true)
            {
                MPI_Recv(&data, 3, MPI_INT, MPI_ANY_SOURCE, REQUEST, MPI_COMM_WORLD, &status);
                Data recv_data;
                recv_data.ID = data[0];
                recv_data.clock = data[1];
                recv_data.weight = data[2];

                // Atomic
                checkClock(recv_data);
                addQueue(recv_data);

                // Send back confirmation message
                MPI_Send(&success, 1, MPI_INT, recv_data.ID, REPLY, MPI_COMM_WORLD);
            }
    });
}

void Skyer::checkClock(const Data& recv_data)
{
    #ifdef DEBUG
        std::cout << m_data.ID << ": checkClock() with " << recv_data.ID << std::endl;
    #endif

    clockMutex.lock();
    m_data.clock = std::max(m_data.clock, recv_data.clock) + 1;
    clockMutex.unlock();
}

void Skyer::sortQueue()
{
    #ifdef DEBUG
        std::cout << m_data.ID << ": sortQueue()" <<  std::endl;
    #endif
    std::sort(queue.begin(), queue.end(), [](const Data &s1, const Data &s2)
    {
         if(s1.clock < s2.clock)
            return true;

        return (s1.clock == s2.clock && s1.ID < s2.ID);
    });
}

void Skyer::addQueue(const Data &data)
{
    #ifdef DEBUG
        std::cout << m_data.ID << ": addQueue() added " << data.ID << std::endl;
    #endif

    queueMutex.lock();
    queue.emplace_back(data);
    queueMutex.unlock();
    sortQueue();
}

void Skyer::deleteQueue(int ID)
{
    #ifdef DEBUG
        std::cout << m_data.ID << ": deleteQueue() removed " << ID << std::endl;
    #endif

    queueMutex.lock();
    auto lambda = [ID](const Data &a){ return a.ID == ID; };
    auto remove = std::remove_if(queue.begin(), queue.end(), lambda);
    queue.erase(remove, queue.end());
    queueMutex.unlock();
    sortQueue();
```

```
}

int Skyer::countAvailableWeight()
{
    int availableWeight = MAX_SKI_LIFT;
    sortQueue();
    for (const auto &data : queue)
    {
        if (data.ID == m_data.ID)
            break;

        availableWeight -= data.weight;
    }
    return availableWeight;
}

void Skyer::waitForSkiLift()
{
    #ifdef DEBUG
        std::cout << m_data.ID << ": waitForSkiLift()" << std::endl;
    #endif
    int availableWeight = countAvailableWeight();
    while (m_data.weight > availableWeight)
    {
        // Waiting
        releaseMutex.lock();
        availableWeight = countAvailableWeight();
    }
    releaseMutex.unlock();
}

void Skyer::updateClock()
{
    #ifdef DEBUG
        std::cout << m_data.ID << ": updateClock()" << std::endl;
    #endif
    clockMutex.lock();
    m_data.clock += 1;
    clockMutex.unlock();
}

void Skyer::localSleep()
{
    #ifdef DEBUG
        std::cout << m_data.ID << ": localSleep()" << std::endl;
    #endif
    int sleepTime = rand() % MAX_SLEEP_TIME + MIN_SLEEP_TIME;
    sleep(sleepTime);
}

void Skyer::requestForSkiLift()
{
    #ifdef DEBUG
        std::cout << m_data.ID << ": requestForSkiLift()" << std::endl;
    #endif

    // Dummy variable to store in-out m_data information
    int data[3];
    data[0] = m_data.ID;
    data[1] = m_data.clock;
    data[2] = m_data.weight;

    // Send request to all but yourself
    for (int process_rank = 0; process_rank < MAX_SKYERS; ++process_rank)
    {
        if (process_rank != m_data.ID)
        {
            MPI_Send(&data, 3, MPI_INT, process_rank, REQUEST, MPI_COMM_WORLD);
        }
    }
}

void Skyer::criticalSection()
{
    #ifdef DEBUG
        std::cout << m_data.ID << ": criticalSection()" << std::endl;
    #endif

    int sleepTime = rand() % MAX_SLEEP_TIME + MIN_SLEEP_TIME;
    std::cout << "Skyer [ID = " << std::setw(3) << m_data.ID << ", weight = " << std::setw(3) << m_data.weight
<< "] rides ski lift for " << sleepTime << " [s]." << std::endl;
```

```
    sleep(sleepTime);
}

void Skyer::releaseSkiLift()
{
    #ifdef DEBUG
        std::cout << m_data.ID << ": releaseSkiLift()" << std::endl;
    #endif

    for (int process_rank = 0; process_rank < MAX_SKYERS; ++process_rank)
    {
        if (process_rank != m_data.ID)
        {
            MPI_Send(&m_data.ID, 1, MPI_INT, process_rank, RELEASE, MPI_COMM_WORLD);
        }
    }
    std::cout << "Skyer [ID = " << std::setw(3) << m_data.ID << ", weight = " << std::setw(3) << m_data.weight
<< "] quits ski lift." << std::endl;
}

void Skyer::waitForConfirm()
{
    #ifdef DEBUG
        std::cout << m_data.ID << ": waitForConfirm()" << std::endl;
    #endif

    MPI_Status status;
    int success;
    for (int process_rank = 0; process_rank < MAX_SKYERS; ++process_rank)
    {
        if (process_rank != m_data.ID)
        {
            MPI_Recv(&success, 1, MPI_INT, process_rank, REPLY, MPI_COMM_WORLD, &status);
        }
    }
}

void Skyer::mainActivity()
{
    while (true)
    {
        // Local section
        localSleep();
        updateClock();

        // Send request to other skyers for ski lift place
        requestForSkiLift();

        // Add to local queue your request
        addQueue({m_data.ID, m_data.clock, m_data.weight});

        // Wait for confirmation from other S-1 skiers
        waitForConfirm();

        // Skyer waits for available ski lift
        waitForSkiLift();

        // Critical section - skyer enters ski lift
        criticalSection();

        // Skyer quits ski lift - sends RELEASE signal to others skyers
        releaseSkiLift();

        // Delete your request from the queue
        deleteQueue(m_data.ID);
    }
}
```

MAIN.CPP

```
#include <iostream>
#include <mpi.h>

#include "Skyer.h"
#include "File.h"
```

```cpp
// mpic++ Main.cpp Skyer.cpp Utility.cpp -o skyers -pthread -D DEBUG -Wall -Wextra -std=c++17
// mpirun -np 4 --hostfile mpi_hosts --map-by node skyers weights.txt
// mpirun -np 4 skyers weights.txt

int main(int argc, char **argv)
{
    // Read from file
    if(argc != 2)
    {
        std::cerr << "Usage: mpirun -n {sample_size} {weights_filename}" << std::endl;
        return EXIT_FAILURE;
    }

    std::string filename(argv[1]);
    std::vector<int> skiersWeights = loadDataFromFile(filename);

    int threads;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &threads);
    if(threads != MPI_THREAD_MULTIPLE) {
        std::cerr << "[MPI_Init_thread] Failed - err: Too little threads" << std::endl;
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    }

    // Predefines
    MAX_SKI_LIFT = 200;
    MAX_SLEEP_TIME = 10;
    MIN_SLEEP_TIME = 1;

    // Shared
    Skyer skyer;
    skyer.m_data.clock = 0;
    srand(time(nullptr) + skyer.m_data.ID);

    // MPI initialize

    // Initialize global size
    MPI_Comm_size(MPI_COMM_WORLD, &MAX_SKYERS);

    // Initialize process own rank
    MPI_Comm_rank(MPI_COMM_WORLD, &skyer.m_data.ID);

    skyer.m_data.weight = skiersWeights.at(skyer.m_data.ID);

    skyer.mainActivity();

    MPI_Finalize();
    return EXIT_SUCCESS;
}
```