Laboratorium Podstawy Przetwarzania Rozproszonego

SPRAWOZDANIE z zadania 4

Nazwisko Imię	album	termin zajęć
Matkowski Mateusz	145432	wtorek 16:40
Mielczarek Damian	145388	

Część I – Algorytm rozwiązania

1. Definicja problemu

Zatoka portowa dysponująca H holownikami; S statków o różnej masie wymagających różnej ilości hi holowników; H < suma(hi). Aby wpłynąć do portu każdy statek musi pozyskać odpowiednią liczbę holowników. Po zacumowaniu statek oddaje holowniki. Port posiada nieskończoną pojemność. Po dokowaniu statek ponownie pozyskuje holowniki, wypływa z portu i zwalnia holowniki. Napisać program dla procesu statku umożliwiający każdemu ze statków wielokrotne korzystanie z portu. Holowniki należy traktować jako zasoby.

2. Założenia przyjętego modelu komunikacji

- asynchroniczny system z wymianą komunikatów
- topologia połączeń: każdy z każdym
- wymagana pojemność kanału: 2 wiadomości w jednym kierunku
- inne wymagane własności sieci komunikacyjnej: kanały typu FIFO

3. Algorytm wzajemnego wykluczania

1. *Każdy statek S_i czeka losową ilość czasu po wpłynięciu do zatoki lub w porcie. (LS)*
2. *Statek S_i wysyła **ŻĄDANIE** (zegar, id, ilość_holowników) o przydział holowników do wszystkich pozostałych S-1 statków i dopisuje je również do swojej kolejki. (LS)*
3. *Statek S_i czeka na odpowiedź (**POTWIERDZENIE**) wszystkich pozostałych S-1 statków. (LS)*
4. *Statek S_i sprawdza w swojej kolejce czy suma zajętych holowników z poprzedzających żądań jest <=H - h_i. (LS)*
    a. *Jeśli dostępna liczba holowników jest niewystarczająco to statek czeka na sygnał **RELEASE**. (LS)*
    b. *Jeśli otrzyma sygnał **RELEASE** to usuwa **ŻĄDANIE** z kolejki i idzie do punktu 4. (LS)*
5. *Statek S_i wpływa do portu lub wypływa do zatoki.(CS)*
6. *Statek S_i zwalnia h_i holowników (wysyła sygnał **RELEASE** do wszystkich statków). (LS)*
7. *Statek S_i usuwa swoje **ŻĄDANIE** ze swojej lokalnej kolejki. (LS)*
8. *Idź do punktu 1. (LS)*

Jeśli statek otrzyma ŻĄDANIE to wysyła sygnał POTWIERDZENIE (potwierdzenie dopisania żądania statku do kolejki) w każdym momencie przetwarzania.
Jeśli statek otrzyma sygnał RELEASE to usuwa ŻĄDANIE ze swojej kolejki.

4. Analiza złożoności komunikacyjnej algorytmu

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

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

*należy wyznaczyć **dokładną** złożoność (nie rząd złożoności), a gdy możliwe są różne przypadki – należy podać złożoność pesymistyczną oraz średnią*

Część II – Implementacja rozwiązania

```
#include <stdio.h>
#include <iostream>
#include <time.h>
#include <algorithm>
#include <vector>
#include <mpi.h>
#include <unistd.h>
#include <pthread.h>

#define REQUEST_TAG 0
#define CONFIRMATION_TAG 1
#define RELEASE_TAG 2

pthread_mutex_t mutex1, mutex2, waitForRelease;

struct RequestVariables
{
    int clock;
    int ID;
    int tugsNeeded;
};

//function compares two values in two objects for sort algorithm
bool compareByClockAndID(const RequestVariables &a, const RequestVariables &b){
    if(a.clock < b.clock)
        return true;
    if(a.clock == b.clock && a.ID < b.ID)
        return true;
    return false;
}
class ShipClass: public RequestVariables{
public:
    //queue is a vector in order to dynamically change queue's length
    std::vector <RequestVariables> RequestsQueue;
    int maxShips;
    int maxTugs;
    unsigned int maxSleepTime;

    void mainLoop();
```

```
    void addToQueue(RequestVariables NewRequest);
    void deleteFromQueue(int deleteID);
    unsigned int countFreeTugs();
    void waitForTugs();
    void printQueue();
};
//function outputs the whole queue on the terminal
void ShipClass::printQueue(){
    std::cout << "(" << ID << ") [";
    for(auto Request : RequestsQueue){
        std::cout << "[" << Request.clock << ", " << Request.ID << ", " << Request.tugsNeeded
<< "], ";
    }
    std::cout << "]\n";
}

void ShipClass::addToQueue(RequestVariables NewRequest){
    deleteFromQueue(NewRequest.ID);
    pthread_mutex_lock(&mutex2);
    RequestsQueue.push_back(NewRequest);
    pthread_mutex_unlock(&mutex2);

    std::sort(RequestsQueue.begin(), RequestsQueue.end(), compareByClockAndID);
}
void ShipClass::deleteFromQueue(int deleteID){
    pthread_mutex_lock(&mutex2);
    for(int i = 0; i < RequestsQueue.size(); i++){
        if(RequestsQueue[i].ID == deleteID){
            RequestsQueue.erase(RequestsQueue.begin() + i);
            i = -1;
        }
    }
    pthread_mutex_unlock(&mutex2);

    std::sort(RequestsQueue.begin(), RequestsQueue.end(), compareByClockAndID);
}
unsigned int ShipClass::countFreeTugs(){
    unsigned int freeTugs = maxTugs;

    std::sort(RequestsQueue.begin(), RequestsQueue.end(), compareByClockAndID);

    for(unsigned int i = 0; i < RequestsQueue.size(); i++){
        if(RequestsQueue[i].ID == ID)
            break;
        freeTugs -= RequestsQueue[i].tugsNeeded;
    }
    return freeTugs;
}

void ShipClass::waitForTugs(){
    unsigned int freeTugs = countFreeTugs();
    unsigned int oldFreeTugs = 500000;
    while(freeTugs < tugsNeeded){
```

```
        if(freeTugs != oldFreeTugs){
            std::cout << "|_______|_______|_____________________________________________\n"
             << "| " << ID << "\t| " << clock  << "\t| Waiting for " << tugsNeeded -
freeTugs;
            if(tugsNeeded - freeTugs != 1)
                std::cout << " tugs.\t\t\t\n";
            else{
                std::cout << " tug.\t\t\t\n";
            }
        }
        pthread_mutex_lock(&waitForRelease);
        freeTugs = countFreeTugs();
        oldFreeTugs = freeTugs;
    }
    pthread_mutex_unlock(&waitForRelease);
}
void ShipClass::mainLoop(){
    int request[3];
    int confirmation;
    unsigned int sleepFor;
    MPI_Status status;
    bool toThePort = true;

    for(;;){
        //Ship waits and does nothing :)
        sleepFor = rand() % maxSleepTime + 1;
        if(toThePort){
            std::cout << "|_______|_______|_____________________________________________\n"
                      << "| " << ID << "\t| " << clock  << "\t| Will sail on the sea for " <<
sleepFor << " seconds.\t\n";
        }
        else{
            std::cout << "|_______|_______|_____________________________________________\n"
                      << "| " << ID << "\t| " << clock  << "\t| Will wait in the port for "
<< sleepFor << " seconds.\t\n";
        }
        sleep(sleepFor);
        //Ship sends a REQUEST signal to S-1 ships
        pthread_mutex_lock(&mutex1);
        clock += 1;
        pthread_mutex_unlock(&mutex1);
        request[0] = clock;
        request[1] = ID;
        request[2] = tugsNeeded;
        for(unsigned int i = 0; i < maxShips; i++){
            if(i == ID)
                continue;
            MPI_Send(&request, 3, MPI_INT, i, REQUEST_TAG, MPI_COMM_WORLD);
        }
        addToQueue({clock, ID, tugsNeeded});
        //Ship waits for the answer from S-1 ships
        for(unsigned int i = 0; i < maxShips; i++){
            if(i == ID)
```

```
                continue;
            MPI_Recv(&confirmation, 1, MPI_INT, i, CONFIRMATION_TAG, MPI_COMM_WORLD,
&status);
        }

        //Ship waits for all needed tugs
        waitForTugs();

        //Ship sails with its tugs
        sleepFor = rand() % maxSleepTime + 1;
        if(toThePort){
            std::cout << "|_______|_______|_____________________________________\n"
                      << "| " << ID << "\t| " << clock  << "\t| Will sail to the port for "
<< sleepFor << " seconds.\t\n";
        }
        else{
            std::cout << "|_______|_______|_____________________________________\n"
                      << "| " << ID << "\t| " << clock  << "\t| Will sail out of the port for
" << sleepFor << " seconds.\t\n";
        }
        sleep(sleepFor);

        //Ship releases all tugs - sends REALSAE signal and deletes its REQUEST from the
queue
        for(unsigned int i = 0; i < maxShips; i++){
            if(i == ID)
                continue;
            MPI_Send(&ID, 1, MPI_INT, i, RELEASE_TAG, MPI_COMM_WORLD);
        }
        toThePort = !toThePort;
        deleteFromQueue(ID);
    }
}

ShipClass Ship;

//function runs in a thread and listens for REQUEST signals
void *receivingRequestThread(void *param) {
    int positive = 1;
    int request[3];
    MPI_Status status;
    for(;;){
        MPI_Recv(&request, 3, MPI_INT, MPI_ANY_SOURCE, REQUEST_TAG, MPI_COMM_WORLD, &status);

        pthread_mutex_lock(&mutex1);
        Ship.clock = std::max(Ship.clock, request[0]) + 1;
        pthread_mutex_unlock(&mutex1);

        Ship.addToQueue({request[0], request[1], request[2]});

        MPI_Send(&positive, 1, MPI_INT, request[1], CONFIRMATION_TAG, MPI_COMM_WORLD);
    }
    pthread_exit(NULL);
```

```
}

//function runs in a thread and listens for RELEASE signals
void *receivingReleaseThread(void *param) {
    int receivedID;
    MPI_Status status;
    for(;;){
        MPI_Recv(&receivedID, 1, MPI_INT, MPI_ANY_SOURCE, RELEASE_TAG, MPI_COMM_WORLD,
&status);

        pthread_mutex_lock(&mutex1);
        Ship.clock += 1;
        pthread_mutex_unlock(&mutex1);

        Ship.deleteFromQueue(receivedID);
        //unlock optimizes tugs counting in waitForTugs()
        pthread_mutex_unlock(&waitForRelease);
    }
    pthread_exit(NULL);
}

int main(int argc, char* argv[]){
    int providedThreads;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &providedThreads);
    if(providedThreads != MPI_THREAD_MULTIPLE){
        std::cout << "Err: Za mało wątków!\n";
        MPI_Finalize();
        return 0;
    }
    MPI_Comm_size(MPI_COMM_WORLD, &Ship.maxShips);
    MPI_Comm_rank(MPI_COMM_WORLD, &Ship.ID);
    Ship.maxTugs = 7;
    Ship.maxSleepTime = 10;

    int tugsList[] = {1, 2, 3, 5, 7};

    srand((Ship.ID+1)*1000);
    Ship.tugsNeeded = tugsList[Ship.ID];
    Ship.clock = 0;

    pthread_t requestThread, releaseThread;

    pthread_create(&requestThread, NULL, receivingRequestThread, NULL);
    pthread_create(&releaseThread, NULL, receivingReleaseThread, NULL);

    Ship.mainLoop();

    MPI_Finalize();
    return 0;
}
```