МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования «Чувашский государственный университет имени И.Н. Ульянова»

Факультет информатики и вычислительной техники

Кафедра вычислительной техники

Отчёт по лабораторным работам по дисциплине  
«Сетевые операционные системы»

Выполнил: студ. гр. ИВТ-41-21

Андреев Д.А.

Чебоксары 2025**Вариант 6.**

На некотором участке земли живут три вида животных: А, В и С (можно заменить на названия реальных животных, я не силен в биологии). Звери А едят зверей В, звери В питаются зверями С, которые, в свою очередь, не прочь полакомиться зверями А. Звери хаотично перемещаются по участку земли (предположим, квадратному, состоящему из ячеек; каждый зверь занимает одну ячейку). Если один зверь попадает в ячейку, занимаемую другим, представляющим другой вид, то один из них съедает другого (смотря кто кем питается). Если в одну ячейку попадают два зверя одного вида, то они производят потомство: еще одного зверя того же вида, после чего один из них, а также потомок переходят на соседние свободные ячейки (два зверя не могут занимать одну ячейку). Если зверь не может найти себе пищу в течении определенного промежутка времени, он погибает. Также звери умирают по старости. Написать программу, моделирующую эту ситуацию. Каждый зверь представляет собой отдельный процесс или поток (в зависимости от реализации). Проанализировать, как благополучие каждого вида животных зависит от двух остальных.

**Решение:**

Решение основано на нескольких гипотезах:

1. Каждое животное слепо и паттерн поведения больше напоминает червей

2. Черви ореинтируются в пространстве слепо и их действие может вызвать раздражитель у другого червя.

3. Если черви находят червя их вида, то найден червь будет опладотворен и спустя время добавит потомство, если он не был убит до..

4. Если червь находит червя другого вида, то он его убивает.

5. Если червь находит иного червя, то тот червь уведомляется что на него наткнулся другой червь и в свою очередь накнувшийс червь будет съеден.

6. Раздражители реализованы через ивенты. Массив ивентв – лисп подобный список состоящий из [HEAD| TAIL], благодоря чему минимизирутеся простой потоков на локах, ведь достаточно прикрепить новую голову при отправке события, либо сохранить ссылку на предыдущую и заменить на NULL.

**Исходный код:**

main.h

#include <stdlib.h>

#include <stdio.h>

#include <limits.h>

#include <time.h>

#include <unistd.h>

#include <pthread.h>

#include <semaphore.h>

struct Cell;

typedef enum {

    FALSE,

    TRUE

} Bool;

//AType < BType < CType < AType

//Род животного

typedef enum {

    NoneType,

    AType,

    BType,

    CType

} Genus;

//позиция на поле

typedef struct {

    int x;

    int y;

} Position;

typedef struct {

    //Лимит животного, сколько он может прожит без еды

    unsigned int maxHungerStrike;

    unsigned int maxAge;

    unsigned int stepTimeSpan;

} Limits;

struct Animal;

// ИДЕЯ, уведомлять животное о том что мы признали что нас съел он

typedef enum {

    // Смерть

    YouAreDead,

    // Отправляется животным которое пытается войти в нашу клетку

    // Но не может нас съесть

    // Или размножится с нами

    // По факту жертва

    KnockToCell,

    // Предложение размножения

    Ywf,

} EventType;

typedef struct {

    EventType eventType;

    struct Animal\* sender;

} Event;

typedef struct EventNode {

    Event event;

    struct EventNode\* next;

} EventNode;

typedef struct Cell {

    struct Animal\*\* ptr;

    Position position;

} Cell;

typedef struct Animal{

    //Возраст, на котором животное в последний раз поело

    unsigned int lastEatTime;

    unsigned int age;

    Bool isDead;

    Genus type;

    Limits\* limits;

    Cell currentCell;

    EventNode\* events;

    pthread\_t threadId;

    pthread\_mutex\_t mutexId;

} Animal;

void sendEvent(Animal\* recipient, Event event);

Animal\* newAnimal(Genus type, Limits\* limits);

void printAnimal(Animal\* animal);

void\* animalLifeCycle(void\* animal);

typedef struct {

    int width;

    int height;

} FieldSize;

typedef struct{

    //Отброшенно из за переполнения

    int\* discardedAnimals;

    //Родилось

    int\* newbornAnimals;

    //Съедено

    int\* eatenAnimals;

    //Умерло по естественным причинам

    int\* dead;

} Statistics;

//Представление поля

typedef struct {

    Animal\*\* pointers;

    Statistics stats;

    FieldSize size;

    // pthread\_mutex\_t transition\_mutex;

} Field;

Field\* newField(int width, int height);

Cell getCell(Field\* field, Position position);

void printField(Field\* field);

Bool setToRandomFreePosition(Animal\* animal, Field\* field);

//По спирали ищет ближайшее свободное место

Bool setToNearestFreePosition(Animal \*animal, Field \*field, Position startPosition);

//Методы жизненного цикла

Position selectNextPosition(Position position, FieldSize fieldSize);

//Не совсем MOVE, скорее какое то действие совершаемое для достижения позиции

void move(Animal\* animal, Field\* field, Position newPosition);

Bool giveBirth(Animal\* animal, Field\* field);

void processEvents(Animal \*animal, Field \*field);

//ЭТИ ФУНКЦИИ ПРОСТО ОТПРАВЛЯЮТ ИВЕНТЫ

void multiply(Animal\* parent1, Animal\* parnet2, Field\* field);

void eatIt(Animal\* predator, Animal\* prey, Field\* field);

typedef struct {

    Animal\* animal;

    Field\* field;

} InitialArgs;

main.c

#include "main.h"

void runAnimalLifeCycle(Animal\* animal, Field\* field);

Animal\*\* createAnimals(Field\* field, int a, int b, int c);

int main(int argc, char const \*argv[])

{

    const int aCount = 10;

    const int bCount = 0;

    const int cCount = 5;

    const int sum = aCount + bCount + cCount;

    setbuf(stdout, 0);

    // Limits\* limits = malloc(sizeof(Limits));

    // limits->maxAge = 30;

    // limits->maxHungerStrike = 10;

    // limits->stepTimeSpan = 5000000;

    Field\* field = newField(20, 10);

    Animal\*\* animals = createAnimals(field, aCount, bCount, cCount);

    printField(field);

    for(int i = 0; i < sum; i++)

    {

        runAnimalLifeCycle(animals[i], field);

    }

    free(animals);

    while (TRUE)

    {

        system("clear");

        printField(field);

    }

    return 0;

}

Field\* newField(int width, int height)

{

    Field\* field = malloc(sizeof(Field));

    field->pointers = calloc(width \* height, sizeof(Animal\*));

    field->stats.dead = malloc(sizeof(int));

    field->stats.discardedAnimals = malloc(sizeof(int));

    field->stats.eatenAnimals = malloc(sizeof(int));

    field->stats.newbornAnimals = malloc(sizeof(int));

    FieldSize size = {width, height};

    field->size = size;

    return field;

}

Animal\*\* createAnimals(Field\* field, int a, int b, int c)

{

    int sum = a+b+c;

    Animal\*\* animals = calloc(sum, sizeof(Animal\*));

    for (int i = 0; i < sum; i++)

    {

        Limits\* limits = malloc(sizeof(Limits));

        limits->maxAge = 30;

        limits->maxHungerStrike = 10;

        limits->stepTimeSpan = 2000000 + rand()%3000000;

        animals[i] = newAnimal(

            AType,

            limits

        );

        setToRandomFreePosition(animals[i], field);

    }

    for (int i = a; i < sum - c; i++)

        animals[i]->type = BType;

    for (int i = a + b; i < sum; i++)

        animals[i]->type = CType;

    return animals;

}

void runAnimalLifeCycle(Animal\* animal, Field\* field)

{

    //а вдруг, вдруг мы попытались родиться вне поля

    if (animal->currentCell.ptr == NULL){

        (\*field->stats.discardedAnimals)++;

        free(animal);

        return;

    }

    (\*field->stats.newbornAnimals)++;

    animal->limits->stepTimeSpan -= rand()%2000000;

    InitialArgs\* args = malloc(sizeof(InitialArgs));

    args->animal = animal;

    args->field = field;

    pthread\_create(&animal->threadId, NULL, &animalLifeCycle, args);

}

void updateCycle(Animal\* animal, Field\* field)

{

    animal->age++;

    if ((animal->age - animal->lastEatTime) >= animal->limits->maxHungerStrike)

    {

        animal->isDead = TRUE;

        (\*field->stats.dead)++;

        return;

    }

    if (animal->age >= animal->limits->maxAge)

    {

        animal->isDead = TRUE;

        (\*field->stats.dead)++;

        return;

    }

}

void\* animalLifeCycle(void \*args)

{

    Animal\* animal = ((InitialArgs\*)args)->animal;

    Field\* field = ((InitialArgs\*)args)->field;

    free(args);

    while (TRUE){

        processEvents(animal, field);

        updateCycle(animal, field);

        usleep(animal->limits->stepTimeSpan);

        processEvents(animal, field);

        if (animal->isDead)

            break;

        Position nextPosition = selectNextPosition(

            animal->currentCell.position,

            field->size

        );

        move(animal, field, nextPosition);

    }

    Cell cell = animal->currentCell;

    if ((\*cell.ptr) == animal){

        (\*cell.ptr) = NULL;

    }

    processEvents(animal, field);

    usleep(animal->limits->stepTimeSpan\*2);

    free(animal);

    return NULL;

}

Animal \*newAnimal(Genus type, Limits \*limits)

{

    Cell cell = {NULL, -1, -1};

    Animal\* animal = malloc(sizeof(Animal));

    animal->age = 0;

    animal->lastEatTime = 0;

    animal->type = type;

    animal->limits = limits;

    animal->currentCell = cell;

    animal->events = NULL;

    pthread\_mutex\_init(&animal->mutexId, NULL);

    return animal;

}

void printAnimal(Animal \*animal)

{

    printf(

        "animal(type: %i, age: %i, lastEatTime: %i, cell.x:%i, cell.y:%i)\n",

        animal->type,

        animal->age,

        animal->lastEatTime,

        animal->currentCell.position.x,

        animal->currentCell.position.y

    );

}

void eatIt(Animal \*predator, Animal \*prey, Field\* field)

{

    (\*field->stats.eatenAnimals)++;

    predator->lastEatTime = predator->age;

    Event event = {YouAreDead, predator};

    sendEvent(prey, event);

}

void multiply(Animal \*mainParent, Animal \*parnet2, Field\* field)

{

    Event event = {Ywf, mainParent};

    sendEvent(parnet2, event);

}

Cell getCell(Field\* field, Position position)

{

    Cell cell = {NULL, -1, -1};

    if (position.x >= field->size.width || position.y >= field->size.height)

        return cell;

    int realPosition = position.x \* field->size.height + position.y;

    cell.ptr = field->pointers + realPosition;

    cell.position = position;

    return cell;

}

void printField(Field \*field)

{

    printf("discarded: %i newborn: %i eaten: %i, dead: %i\n",

        \*field->stats.discardedAnimals,

        \*field->stats.newbornAnimals,

        \*field->stats.eatenAnimals,

        \*field->stats.dead

    );

    for (int j = 0; j < field->size.height; j++)

    {

        for (int i = 0; i < field->size.width; i++)

        {

            Position position = {i, j};

            Cell cell = getCell(field, position);

            Animal\* animal = (\*cell.ptr);

            int type = 0;

            if (animal != NULL)

                type = animal->type;

            switch (type)

            {

                case 1:

                    printf("\033[34m%2i\033[0m", animal->age);

                    break;

                case 2:

                    printf("\033[35m%2i\033[0m", animal->age);

                    break;

                case 3:

                    printf("\033[32m%2i\033[0m", animal->age);

                    break;

                default:

                    printf("░░");

            }

        }

        printf("\n");

    }

}

Bool setToRandomFreePosition(Animal \*animal, Field \*field)

{

    Position startPosition = {

        rand() % field->size.width,

        rand() % field->size.height

    };

    return setToNearestFreePosition(animal, field, startPosition);

}

Bool setToNearestFreePosition(Animal \*animal, Field \*field, Position startPosition)

{

    int generation = 0;

    int lastGeneration = -1;

    while (generation < field->size.height || generation < field->size.width){

        for (int i = -generation; i <= generation; i += 1)

        {

            Position newPosition = { startPosition.x + i, 0};

            if (newPosition.x < 0)

                continue;

            if (newPosition.x >= field->size.width)

                break;

            for (int j = -generation; j <= generation; j += 1)

            {

                if (abs(j) <= lastGeneration && abs(i) < lastGeneration){

                    j = abs(j);

                    continue;

                }

                newPosition.y = startPosition.y + j;

                if (newPosition.y < 0)

                    continue;

                if (newPosition.y >= field->size.height)

                    break;

                Cell cell = getCell(field, newPosition);

                if ((\*cell.ptr) != NULL)

                    continue;

                (\*cell.ptr) = animal;

                animal->currentCell = cell;

                return TRUE;

            }

        }

        lastGeneration = generation;

        generation++;

    }

    return FALSE;

}

void acceptEvent(Animal\* animal, EventNode\* eventNode, Field\* field)

{

    if (eventNode == NULL)

        return;

    EventNode\* temp = eventNode->next;

    eventNode->next = NULL;

    acceptEvent(animal, temp, field);

    if (animal->isDead)

    {

        free(eventNode);

        return;

    }

    Event event = eventNode->event;

    switch (event.eventType)

    {

        case YouAreDead:

            /\* code \*/

            animal->isDead = TRUE;

            break;

        case KnockToCell:

            eatIt(animal, event.sender, field);

            break;

        case Ywf:

            giveBirth(animal, field);

            break;

        default:

            break;

    }

    free(eventNode);

}

void processEvents(Animal \*animal, Field \*field){

    pthread\_mutex\_lock(&animal->mutexId);

    EventNode\* events = animal->events;

    animal->events = NULL;

    pthread\_mutex\_unlock(&animal->mutexId);

    acceptEvent(animal, events, field);

}

void sendEvent(Animal \*recipient, Event event)

{

    if (recipient->isDead)

        return;

    pthread\_mutex\_lock(&recipient->mutexId);

    if (recipient->isDead)

    {

        pthread\_mutex\_unlock(&recipient->mutexId);

        return;

    }

    EventNode\* eventNode = malloc(sizeof(EventNode));

    eventNode->event = event;

    eventNode->next = recipient->events;

    recipient->events = eventNode;

    pthread\_mutex\_unlock(&recipient->mutexId);

}

Position selectNextPosition(Position position, FieldSize fieldSize)

{

    const int offsetsCount = 6;

    Position offsets[] = {{-1, -1}, {-1, 0}, {0, -1}, {1, 0}, {0, 1}, {1, 1}};

    int initial = rand() % offsetsCount - 1;

    int i = initial;

    while(TRUE)

    {

        Position newPosition = {

            position.x + offsets[i].x,

            position.y + offsets[i].y

        };

        i = ( i + 1 ) % offsetsCount;

        if (i == initial)

            return offsets[0];

        if (newPosition.x < 0)

            continue;

        if (newPosition.y < 0)

            continue;

        if (newPosition.x >= fieldSize.width)

            continue;

        if (newPosition.y >= fieldSize.height)

            continue;

        return newPosition;

    }

    return offsets[0];

}

void doAction(Animal\* animal, Animal\* cellOwner, Field\* field)

{

    EventType eventType;

    if (cellOwner == animal)

        return;

    if (cellOwner->type == animal->type)

    {

        multiply(animal, cellOwner, field);

    }

    else if (cellOwner->type == ((animal->type - 1 + 4) % 4))

    {

        eatIt(animal, cellOwner, field);

    }

    else

    {

        Event event = {

            KnockToCell,

            animal

        };

        sendEvent(cellOwner, event);

    }

}

void move(Animal \*animal, Field \*field, Position newPosition)

{

    Cell newCell = getCell(field, newPosition);

    Animal\* cellOwner = (\*newCell.ptr);

    if (cellOwner != NULL)

    {

        doAction(animal, cellOwner, field);

        return;

    }

    (\*newCell.ptr) = animal;

    Cell oldCell = animal->currentCell;

    animal->currentCell = newCell;

    usleep(animal->limits->stepTimeSpan);

    (\*oldCell.ptr) = NULL;

}

Bool giveBirth(Animal \*animal, Field \*field)

{

    Animal\* child = newAnimal(animal->type, animal->limits);

    Bool hasCell = setToRandomFreePosition(child, field);

    runAnimalLifeCycle(child, field);

    return hasCell;

}