Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работ №2 по курсу

«Операционные системы»

Группа: М80 – 206Б-18
Студент: Касимов М.М.
Преподаватель: Соколов А.А.
Оценка: _____
Дата: _____

Москва, 2019.

Содержание

Постановка задачи.

Произвести распараллеленный поиск по дереву общего вида. Первоначальные данные задаются в файле в виде простых команд (например, "+ 1 /" добавить элемент к корню дерева, "+ 2 1/3/" добавить элемент 2 по пути 1->3 в дереве).

Общие сведения о программе

Программа компилируется с ключом - lpthread. Используются заголовочные файлы stdio.h, unistd.h, stdbool.h, stdlib.h, pthread.h, queue.h (очередь, которую я написал для данной работы). В программе используются следующие системные вызовы:

1. **pthread_mutex_lock** – для захвата мьютекса потоком
2. **pthread_create** – для создания потока
3. **pthread_join** – для ожидания завершения потока
4. **pthread_mutex_unlock** – для освобождения мьютекса потоком
5. **pthread_exit** – для завершения потока

Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Создание общего дерева используя данные из файла.
2. Используя системный вызов создать «дочерний» поток.
3. В функции поиска для каждого узла конкретного уровня дерева создавать потоки для поиска.
4. Узлы, которым не хватило потоков, мы помещаем в очередь и производим поиск в ширину.
5. Когда мы найдем наш элемент в дереве, мы изменим булево поле в структуре на true.

Основные файлы программы.

Файл queue.h

```
#ifndef OS_LAB_3_QUEUE_H
#define OS_LAB_3_QUEUE_H

#include <stdbool.h>
#include <stdlib.h>

struct node {
        int val;
       struct node * son;
       struct node * bro;
};

typedef struct node Tree;


typedef struct QueueItem queue_item;
struct QueueItem {
   struct QueueItem* next;
   struct QueueItem* prev;
   Tree* value;
};

typedef struct Queue queue;
struct Queue {
   queue_item* head;
   queue_item* tail;
   size_t size;
};
```

Файл queue.c

```
#include "queue.h"
void q_init(queue* q) {
   q->head = NULL;
   q->tail = NULL;
```

```
    q->size = 0;
}

Tree* q_top(queue* q) {
    return q->head->value;
}

Tree* q_pop(queue* q) {

    Tree* temp = q_top(q);
    queue_item* ptr_to_free = q->head;
    q->head = q->head->next;
    if (q->head == NULL) {
        q->tail = NULL;
    }
    free(ptr_to_free);
    q->size--;
    return temp;
}

void q_push(queue* q, Tree* elem) {
    queue_item* new_elem = malloc(sizeof(queue_item));
    new_elem->value = elem;
    new_elem->next = 0;
    if (q->head == NULL) {
        q->head = new_elem;
        q->tail = new_elem;
        new_elem->prev = 0;
    } else {
        q->tail->next = new_elem;
        new_elem->prev = q->tail;
        q->tail = new_elem;
    }
    q->size++;
}

bool q_empty(queue* q) {
    return q->head == NULL;
```

```
}

size_t q_size(queue* q) {
    return q->size;
}

void q_destroy(queue* q) {
    queue_item* start = q->head;
    while (start != NULL) {
        queue_item* next = start->next;
        free(start);
        start = next;
    }
    q->head = NULL;
    q->tail = NULL;
    q->size = 0;
}
```

Файл main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include <pthread.h>
#include "queue.h"




typedef struct ThreadParams thread_params;

void pre_check(bool* flag) {
    FILE* file = fopen("in.txt","r");
    char temp = fgetc(file);
    putchar(temp);
```

```
    while (temp != EOF) {
        if ( temp < '0' && temp != '+' && temp != '/' && temp != ' ' && temp != '\n' && temp != EOF ||
temp != '\n'&& temp != EOF && temp > '9' && temp != '+' && temp != '/' && temp != ' '  ) {
            *flag = true;

            fclose(file);
            return;
        }
        temp = fgetc(file);
        if (temp != EOF)
            putchar(temp);

    }
    fclose(file);
}

Tree* Tree_create(int v) {
    Tree* tree = (Tree *) malloc(sizeof(Tree));
    if (!tree){
        printf("Out of memory\n");
        exit(0);
    }
    tree->val = v;
    tree->son = NULL;
    tree->bro = NULL;
    return tree;
}

Tree* find_in_son (Tree* root,int v) {
    if (root ->son == NULL)
        return root;
    Tree* tmp = root->son;
    while (tmp->val != v) {
        tmp = tmp->bro;
```

```
    }
    return tmp;
}


struct ThreadParams {
    Tree* root;
    int num;
    bool* found;
    int* thread_count;
    pthread_mutex_t* count_mutex;
};



void* tree_find(void* arg) {
    thread_params* params = (thread_params*) arg;
    bool* found = params->found;
    int num = params->num;
    Tree* root = params->root;
    int* thread_count = params->thread_count;
    pthread_mutex_t* count_mutex = params->count_mutex;

    if (root == NULL){
        pthread_exit(NULL);
    }

    if (root->val == num) {
        *(params->found) = true;
        pthread_exit(NULL);
    }

    if (root->son == NULL) {
        pthread_exit(NULL);
```

```
}

int son_count = 0;
Tree* son = root->son;
while(son != NULL) {
   if (son->val == num) {
         *(params->found) = true;
         pthread_exit(NULL);
   }
   son_count++;
   son = son->bro;

}
son = root->son;
Tree* sons[son_count];
int counter = 0;
while(son != NULL) {
   sons[counter] = son;
   counter++;
   son = son->bro;
}
pthread_mutex_lock(count_mutex);
int create_threads = (*thread_count > son_count) ? son_count : *thread_count;
(*thread_count) -= create_threads;
pthread_mutex_unlock(count_mutex);
pthread_t threads[create_threads];
thread_params* new_params = malloc(sizeof(thread_params) * create_threads);
for (int i = 0; i < create_threads; ++i) {
   new_params[i] = *params;
   new_params[i].root = sons[i];
   pthread_create(&threads[i], NULL, tree_find, &new_params[i]);
}
queue q;
q_init(&q);
```

```
    for (int i = create_threads; i < son_count; ++i) {
        q_push(&q, sons[i]);
    }
    while (!q_empty(&q)) {
        Tree* item = q_pop(&q);
        if (item->val == num) {
            *found = true;
            break;
        }
        Tree* item_son = item->son;
        while (item_son != NULL) {
            q_push(&q, item_son);
            item_son = item_son->bro;
        }
    }
    for (int i = 0; i < create_threads; ++i) {
        pthread_join(threads[i], NULL);
    }
    pthread_mutex_lock(count_mutex);
    (*thread_count) += create_threads;
    pthread_mutex_unlock(count_mutex);
    free(new_params);
    q_destroy(&q);
    pthread_exit(NULL);
}

void pars(Tree** root,char* filename) {
    FILE* in = fopen (filename,"r");
    char tmp;
    int status;
    while ((tmp = fgetc(in)) != EOF) {
        if (tmp == ' ' || tmp == '\n') {
            tmp = fgetc(in);
            continue;
```

```
}
if (tmp == '+') {
        Tree** ptr = root;
        int v = 0,trash = 0,n = 0;
        while ((tmp = fgetc(in)) != ' ') {
           v *= 10;
           v += tmp - '0';
        }
        //fgetc(in);
        while ((tmp = fgetc(in)) != '\n') {
           if ( tmp !=  '/') {
              n *= 10;
              n += tmp - '0';
           }
           else {
              while (*ptr != NULL) {
                 if ((*ptr)->val == n)
                    ptr = &(*ptr) -> son;
                 else {
                    ptr = &(*ptr)->bro;
                    if ((*ptr)->val == n)
                       ptr = &(*ptr) ->son;
                 }
                 break;
              }
              n = 0;
           }
        }

        while (*ptr != NULL)
           ptr = &(*ptr) -> bro;
        *ptr = (Tree*) malloc(sizeof(Tree));
        (*ptr)->val = v;
        (*ptr)->son = NULL;
```

```
            (*ptr)->bro = NULL;
        }
    }
}

void tabs(int n) {
    printf("%d: ",n);
    for (int i = 0; i < n; ++i)
        printf("\t");
}

void print_tree(Tree* root,int lvl) {
    if (!root)
        return;
    Tree* temp = root;
    while (temp != NULL) {
        tabs(lvl);
        printf("%d\n",temp -> val);
        print_tree(temp -> son,lvl + 1);
        temp = temp->bro;
    }
}



int main(int argc,char** argv)
{
    bool flag = false;
    Tree * root = NULL;
    if (flag) {
        printf("error\n");
        return 0;
    }
    pars(&root,argv[1]);
```

```
    print_tree(root,0);
    printf("enter key to find\n");
    int num;
    scanf("%d",&num);
    bool* found = malloc(sizeof(bool));
    *found = false;
    pthread_mutex_t* count_mutex = malloc(sizeof(pthread_mutex_t));
    pthread_mutex_init(count_mutex, NULL);
    printf("enter max thread count\n");
    int* thread_count = malloc(sizeof(int));
    scanf("%d",thread_count);
    thread_params params = {
        .found = found,
        .num = num,
        .root = root,
        .thread_count = thread_count,
        .count_mutex = count_mutex
    };
    pthread_t first_thread;
    pthread_create(&first_thread, NULL, tree_find, &params);
    pthread_join(first_thread, NULL);
    printf("Node is%s found", *found ? "" : " not");
    printf("\n");
    return 0;
}
```

Демонстрация работы программы.

```
magomed@DESKTOP-PG5DLO1:~/os3$ cat in2.txt
+3 /
+4 3/
+5 3/
+11 3/4/
+777 3/4/11/
+42 3/4/11/
magomed@DESKTOP-PG5DLO1:~/os3$ ./a.out in2.txt
0: 3
1:     4
2:          11
3:               777
3:               42
1:     5
enter key to find
11
enter max thread count
3
Node is found
magomed@DESKTOP-PG5DLO1:~/os3$ ./a.out in2.txt
0: 3
1:     4
2:          11
3:               777
3:               42
1:     5
enter key to find
42
enter max thread count
1
Node is found
magomed@DESKTOP-PG5DLO1:~/os3$ ./a.out in2.txt
0: 3
1:     4
2:          11
3:               777
3:               42
1:     5
enter key to find
14
enter max thread count
1000
```

```
Node is not found
magomed@DESKTOP-PG5DLO1:~/os3$ ./a.out in2.txt
0: 3
1:     4
2:          11
3:               777
3:               42
1:     5
enter key to find
90
enter max thread count
1
Node is not found
```

Вывод.

Я научился создавать потоки и взаимодействовать с ними. Узнал что такое мютекс и как работают потоки в ОС Linux. Улучшил навыки программирования на языке программирования Си..