Міністерство освіти і науки України

Харківський національний університет імені В. Н. Каразіна

Факультет комп'ютерних наук

Лабораторна робота №9

з навчальної дисципліни

«Операційні системи»

Виконав:
студент групи КУ-31
Нечипоренко Д.І.

Харків – 2022

Завдання

Задание №1 (Состояние гонки)

Напишите программу, которая показывает возможность повреждения данных (переменная длинного целого типа с нулевым начальным значением) несколькими одновременно работающими потоками (демонстрирует поведение, описывающееся понятием *race conditions*). Программа запускает некоторое количество пар потоков-потомков. Поток из одной части этой пары заданное количество раз увеличивает значение переменной на 1 , поток из другой части пары такое же количество раз уменьшает значение переменной на 1 . Основной поток ожидает завершения работы всех потоков — потомков и выводит финальное значение изменяемой переменной. Количество потоков и количество операций, которые выполняет каждый поток, передаются в программу с помощью коротких опций. Если какая-то опция не задана, то предусмотреть для нее значение по умолчанию. Программа, должна заданное количество раз выполнить эти действия и каждый раз вывести то финальное значение переменной, какое получится в результате каждого запуска.

Проанализируйте поведение программы в зависимости от значения параметров.

Задание №2 (Мьютексы)

Напишите программу, которая находит численное значение определенного интеграла $\int_a^b f(x)\,\mathrm{d}\,x$ методом средних прямоугольников с помощью заданного количества параллельно работающих потоков. Количество потоков p , с помощью которых проводятся расчеты, функция вычисления интеграла получает при запуске программы через параметры командной строки.

Замечание: Для такого многопотокового режима работы программы можно воспользоваться свойством аддитивности интеграла:

$$\int_a^b f(x)\,\mathrm{d}\,x=\sum_{i=0}^{p-1}\int_{a_i}^{b_i} f(x)\,\mathrm{d}\,x,$$

где $a_i=a+i\cdot h$, $b_i=a_i+h$, $h=(b-a)/p$.

Таким образом, участок интегрирования $[a,b]$ разбивается на части, число которых совпадает с количеством работающих потоков. Каждый поток вычисляет свою часть интеграла (интеграл на своем участке $[a_i,b_i]$) и прибавляет его к переменной, в которой «накапливается» результат. Для исключения несанкционированного доступа со стороны работающих потоков к «накапливающей» переменной необходимо защитить ее с помощью мьютекса.

Задание №3 (Блокировки чтения и записи)

Существует массив, количество элементов которого задается с помощью короткой опции при запуске программы. Если при запуске значение не указано, то создается массив из заданного по умолчанию количества элементов. Заданное количество присоединенных потоков-писателей записывают в него информацию (псевдослучайные числа из требуемого диапазона в элементы, выбранные псевдослучайным образом), заданное количество присоединенных потоков-читателей (большее, чем писателей) считывает информацию (сохраненные в массиве числа из элементов, выбранных псевдослучайным образом). Отсоединенный поток с заданной периодичностью выводит состояние массива в стандартный поток вывода. Для удобства работы можно организовать необходимые случайные задержки. Согласованную работу системы (синхронизированный доступ к массиву) осуществите с помощью блокировок чтения-записи.

Код:

Task 1

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

long int val = 0;

void *increment(void *arg) {
    long i, num = *((long *)arg);
    for (i = 0; i < num; i++) {
        val += 1;
    }
    return NULL;
}

void *decrement(void *arg) {
    long i, num = *((long *)arg);
```

```
    for (i = 0; i < num; i++) {
        val -= 1;
    }
    return NULL;
}

int main() {
    pthread_t id1, id2;
    long n;
    printf("\tTest Task1 Conditions\n");
    printf("n: ");
    scanf("%ld", &n);
    printf("Let do %ld pair operations\n", n);
    printf("Before work num = %ld\n", val);
    printf("\tStart ScoreThreads...\n");

    if (pthread_create(&id1, NULL, increment, (void*)&n)) {
        fprintf(stderr, "ERROR!!! Cannot Create Thread\n");
        exit(EXIT_FAILURE);
    }
    if (pthread_create(&id2, NULL, decrement, (void*)&n)) {
        fprintf(stderr, "ERROR!!! Cannot Create Thread\n");
        exit(EXIT_FAILURE);
    }

    pthread_join(id1, NULL);
    pthread_join(id2, NULL);
```

```
    printf("After work num = %ld\n", val);

    return 0;
}
```

Task 2

Main

```
#include <stdio.h>
#include <stdlib.h>

#define ONE_THREAD
#undef ONE_THREAD

#ifdef ONE_THREAD
#include "integr.h"
#else
#include "my_threads.h"
#endif

double fun(double x) {
   return 4.0 - x*x;
}

int main(void) {
   double res;
   printf("Task 1. Integral Calculation\n");
#ifdef ONE_THREAD
```

```
    printf("Test Application without Threads\n");
        res = NIntegrate(fun, 0.0, 2.0, 1.0e-6);
#else
    printf("Threaded Application\n");
    {
        TARG arg = {5, 0.0, 2.0, 1.0e-6, fun};
        res = Thread_NIntegrate(&arg);
    }
#endif
    printf("\tRes: %g\n", res);

    return EXIT_SUCCESS;
}
```

Threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#include "my_threads.h"
#include "integr.h"


typedef struct {
    double * res;
    pthread_mutex_t *m;
```

```
    TARG arg;
} THREAD_ARG;



static void * integr_thread(void * arg) {
    THREAD_ARG data = *(THREAD_ARG*) arg;
    double res;
#ifdef DEBUG
    printf("Score Thread %d is started\n", data.arg.n);
#endif
    res = NIntegrate(data.arg.f, data.arg.a, data.arg.b, data.arg.eps);
    pthread_mutex_lock(data.m);
    *data.res += res;
#ifdef DEBUG
    printf("Score Thread %d is in critical section\n", data.arg.n);
#endif
    pthread_mutex_unlock(data.m);
#ifdef DEBUG
    printf("Score Thread %d is finished\n", data.arg.n);
#endif
    return NULL;
}



double Thread_NIntegrate(const TARG * arg) {
    int i;
    double h, result;
```

```
    pthread_mutex_t total_res_mutex;
    pthread_t *thread;
    THREAD_ARG *arr;



    result = 0;
    h = (arg->b - arg->a) / arg->n;
    if (pthread_mutex_init(&total_res_mutex, NULL)) {
        fprintf(stderr, "ERROR! %s\n", "Cannot initialize mutex!");
        exit(EXIT_FAILURE);
    }
    thread = (pthread_t*)malloc(arg->n * sizeof(pthread_t));
    arr = (THREAD_ARG*)malloc(arg->n * sizeof(THREAD_ARG));
    if ((thread == NULL) || (arr == NULL)) {
        fprintf(stderr, "Allocation Memory Error\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < arg->n; i++) {
        arr[i].res = &result;
        arr[i].m = &total_res_mutex;
        arr[i].arg.n = i;
        arr[i].arg.a = arg->a + i*h;
        arr[i].arg.b = arr[i].arg.a + h;
        arr[i].arg.eps = arg->eps;
        arr[i].arg.f = arg->f;
#ifdef DEBUG
```

```
            printf("Interval [%g, %g]\n", arr[i].arg.a, arr[i].arg.b);
        #endif
            if (pthread_create(&thread[i], NULL, &integr_thread, &arr[i]) != 0) {
                fprintf(stderr, "Score Thread %d Creation Error\n", i);
                exit(EXIT_FAILURE);
            }
        }

        for (i = 0; i < arg->n; i++) {
            if (pthread_join(thread[i], NULL) != 0) {
                fprintf(stderr, "Score Thread %d Waiting Error\n", i);
                exit(EXIT_FAILURE);
            }
        }

        pthread_mutex_destroy(&total_res_mutex);
        free(thread); free(arr);
        return result;
    }
```

Integer

```
#include "integr.h"
#include <stdio.h>
#include <math.h>

double integrate(double (*f)(double x), double a, double b, unsigned long n) {
    double res, h;
```

```
    unsigned long i;
    h = (b - a) / n;
    res = 0;
    for (i = 0; i < n; i++) {
        res += f(a + (i + 0.5)*h);
    }
    res *= h;
    return res;
}

double NIntegrate(double (*f)(double x), double a, double b, double eps) {
    double i0, i1, error;
    unsigned long n = 10;

    i0 = integrate(f, a, b, n);
    do {
        n *= 2;
        i1 = integrate(f, a, b, n);
        error = fabs(i0 - i1);
        i0 = i1;
    } while(error >= eps);

    return i1;
}
```

Task 3

Main

```
#include <stdio.h>
#include <stdlib.h>

#include "task.h"
//#include "tarray.h"

int main(void) {
    int array_size = 7;
    printf("Task3. Read/Write Lock\n");
    {
        TPARAM init;
        TARRAY arr;
        tarray_init(&arr, array_size);
        init.arr = &arr;
        init.num_readers = 6;
        init.num_writers = 3;
        init.num_reps = 10;
        task_solution(&init);
        tarray_dest(&arr);
    }

    printf("\n\nNext Session...\n");
    array_size = 5;
    {
        TPARAM init;
        TARRAY arr;
        tarray_init(&arr, array_size);
        init.arr = &arr;
        init.num_readers = 3;
        init.num_writers = 2;
        init.num_reps = 5;
        task_solution(&init);
```

```
        tarray_dest(&arr);
    }

    return EXIT_SUCCESS;
}
```

Task

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#include "task.h"
#include "threads.h"

typedef struct {
    pthread_t * thread;
    TARG * arg;
} MAKE_RES;

static pthread_t make_detached(TARRAY * arr);
static MAKE_RES * make_writers(TARG * targ);
static MAKE_RES * make_readers(TARG * targ);


void task_solution(TPARAM * init) {
    pthread_t thread;
    TARG arg_writer, arg_reader;
    MAKE_RES * writer;
    MAKE_RES * reader;
    int i;
```

```
printf("Work with %d writer threads\n", init->num_writers);
printf("Work with %d reader threads\n", init->num_readers);
printf("All threads make %d reaps\n", init->num_reps);
printf("Initial array: ");
tarray_print(init->arr, &stdout);
printf("\n");
thread = make_detached(init->arr);
//sleep(5);
arg_writer.arr = init->arr;
arg_writer.num_reps = init->num_reps;
arg_writer.num_thread = init->num_writers;
writer = make_writers(&arg_writer);
arg_reader.arr = init->arr;
arg_reader.num_reps = init->num_reps;
arg_reader.num_thread = init->num_readers;
reader = make_readers(&arg_reader);

for (i = 0; i < init->num_writers; i++) {
   if (pthread_join(writer->thread[i], NULL) != 0) {
      fprintf(stderr, "Writer Thread %d Waiting Error\n", i);
      exit(EXIT_FAILURE);
   }
}
for (i = 0; i < init->num_readers; i++) {
   if (pthread_join(reader->thread[i], NULL) != 0) {
      fprintf(stderr, "Reader Thread %d Waiting Error\n", i);
      exit(EXIT_FAILURE);
   }
}

free(writer->thread);  free(writer->arg);  free(writer);
free(reader->thread);  free(reader->arg);  free(reader);
```

```
    if(pthread_cancel(thread)) {
        fprintf(stderr, "ERROR! Cannot stop detached thread!\n");
        exit(EXIT_FAILURE);
    }

    printf("Result array : ");
    tarray_print(init->arr, &stdout);
    printf("\n");
    sleep(5);
}


static pthread_t make_detached(TARRAY * arr) {
    pthread_attr_t attr;
    pthread_t thread;
    int a;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    if (!pthread_attr_getdetachstate(&attr, &a)) {
        printf("%d\t%d\t", a, PTHREAD_CREATE_DETACHED);
        if (a == PTHREAD_CREATE_DETACHED) {
            printf("Really detached\n");
        }
    }
    if (pthread_create(&thread, &attr, &detach_thread, (void*)arr)) {
        fprintf(stderr, "Error while Creation Detached Thread\n");
        exit(EXIT_FAILURE);
    }
    pthread_attr_destroy(&attr);
```

```
    return thread;
}


static MAKE_RES * make_writers(TARG * targ) {
    MAKE_RES * res;
    TARG * arg;
    int i;

    res = (MAKE_RES *) malloc(sizeof(MAKE_RES));
    if (!res) {
        fprintf(stderr, "Allocation memory error\n");
        exit(EXIT_FAILURE);
    }
    res->thread = (pthread_t *) calloc(targ->num_thread, sizeof(pthread_t));
    res->arg = (TARG *) calloc(targ->num_thread, sizeof(TARG));
    if ((res->thread == NULL) || (res->arg == NULL)) {
        fprintf(stderr, "Allocation memory error\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < targ->num_thread; i++) {
        res->arg[i].num_reps = targ->num_reps;
        res->arg[i].num_thread = i;
        res->arg[i].arr = targ->arr;
        if (pthread_create(&(res->thread[i]), NULL, &write_thread, &(res->arg[i]))) {
            fprintf(stderr, "Writer Thread %d Creation Error\n", i);
            exit(EXIT_FAILURE);
        }
    }

    return res;
```

```
}


static MAKE_RES * make_readers(TARG * targ) {
    MAKE_RES * res;
    TARG * arg;
    int i;

    res = (MAKE_RES *) malloc(sizeof(MAKE_RES));
    if (!res) {
        fprintf(stderr, "Allocation memory error\n");
        exit(EXIT_FAILURE);
    }
    res->thread = (pthread_t *) calloc(targ->num_thread, sizeof(pthread_t));
    res->arg = (TARG *) calloc(targ->num_thread, sizeof(TARG));
    if ((res->thread == NULL) || (res->arg == NULL)) {
        fprintf(stderr, "Allocation memory error\n");
        exit(EXIT_FAILURE);
    }

    for (i = 0; i < targ->num_thread; i++) {
        res->arg[i].num_reps = targ->num_reps;
        res->arg[i].num_thread = i;
        res->arg[i].arr = targ->arr;
        if (pthread_create(&(res->thread[i]), NULL, &read_thread, &(res->arg[i]))) {
            fprintf(stderr, "Reader Thread %d Creation Error\n", i);
            exit(EXIT_FAILURE);
        }
    }

    return res;
}
```

My threads

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#include "threads.h"
#include "tarray.h"

void * write_thread(void * arg) {
    TARG * targ = (TARG *)arg;
    int k, ind;
    double val;
    printf("\tWriter Thread %d is started\n", targ->num_thread);
    sleep(targ->num_thread);
    srand(time(NULL));
    for (k = 0; k < targ->num_reps; k++) {
        ind = rand() % targ->arr->size;
        val = (double)rand() / RAND_MAX;
        tarray_set(targ->arr, ind, val);
        printf("\tWriter Thread No %d, res[%d] = %g\n", targ->num_thread, ind, val);
        sleep(ind);
    }
    printf("\tWriter Thread %d is stopped\n", targ->num_thread);
    return NULL;
}

void * read_thread(void * arg) {
    TARG * targ = (TARG *)arg;
    int k, ind;
    double res;
```

```
    printf("\t\tReader Thread %d is started\n", targ->num_thread);
    sleep(targ->num_thread);
    srand(time(NULL));
    for (k = 0; k < targ->num_reps; k++) {
        ind = rand() % targ->arr->size;
        res = tarray_get(targ->arr, ind);
        printf("\t\tReader Thread No %d, res[%d] = %g\n", targ->num_thread, ind, res);
        sleep(ind);
    }
    printf("\t\tReader Thread %d is stopped\n", targ->num_thread);
    return NULL;
}


void * detach_thread(void * arg) {
    TARRAY arr = *(TARRAY*)arg;

    if (pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL)) {
        fprintf(stderr, "Cannot change cancel state\n");
        exit(EXIT_FAILURE);
    }
    if (pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL)) {
        fprintf(stderr, "Cannot change cancel type\n");
        exit(EXIT_FAILURE);
    }
    printf("\tDetached Thread is Started\n");
    while (1) {
        printf("Current state:");
        tarray_print(&arr, &stdout);
        printf("\n");
        sleep(1);
    }
```

```
    printf("\tDetached Thread is Stopped\n");
    return NULL;
}
```

Результат:

Task 1

```
gcc main.c -lpthread -o main.out
./main.out
        Test Task1 Conditions
n: 1
Let do 1 pair operations
Before work num = 0
        Start ScoreThreads...
After work num = 0
```

Task 2

```
gcc main.c  integr.c my_threads.c -lpthread -o main.out
./main.out
Task 1. Integral Calculation
Threaded Application
        Res: 5.33333
```

Task 3

```
gcc main.c tarray.c task.c threads.c -lpthread  -o ./main.out
./main.out
Task3. Read/Write Lock
Work with 3 writer threads
Work with 6 reader threads
All threads make 10 reaps
Initial array:    0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
1       1       Really detached
        Writer Thread 1 is started
        Writer Thread 2 is started
                Reader Thread 0 is started
                Reader Thread 1 is started
                Reader Thread 2 is started
                Reader Thread 3 is started
                Reader Thread 4 is started
                Reader Thread 5 is started
        Writer Thread 0 is started
        Detached Thread is Started
Current state:   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
                Reader Thread No 0, res[5] = 0
        Writer Thread No 0, res[5] = 0.645633
        Writer Thread No 1, res[5] = 0.554754
                Reader Thread No 1, res[5] = 0.554754
```

```
Current state:   0.0000   0.0877   0.0000   0.9744   0.0000   0.3367   0.0000
                Reader Thread No 2, res[6] = 0
        Writer Thread No 1, res[2] = 0.100105
                Reader Thread No 1, res[6] = 0
                Reader Thread No 3, res[2] = 0.100105
                Reader Thread No 4, res[4] = 0
        Writer Thread No 2, res[3] = 0.602579
Current state:   0.0000   0.0877   0.1001   0.6026   0.0000   0.3367   0.0000
Current state:   0.0000   0.0877   0.1001   0.6026   0.0000   0.3367   0.0000
                Reader Thread No 3, res[3] = 0.602579
        Writer Thread No 1, res[5] = 0.375555
Current state:   0.0000   0.0877   0.1001   0.6026   0.0000   0.3756   0.0000
                Reader Thread No 0, res[6] = 0
        Writer Thread No 2, res[2] = 0.311459
Current state:   0.0000   0.0877   0.3115   0.6026   0.0000   0.3756   0.0000
                Reader Thread No 4, res[1] = 0.0876856
        Writer Thread No 0, res[6] = 0.352355
Current state:   0.0000   0.0877   0.3115   0.6026   0.0000   0.3756   0.3524
                Reader Thread No 5, res[3] = 0.602579
        Writer Thread No 2, res[4] = 0.120235
                Reader Thread No 2, res[6] = 0.352355
                Reader Thread No 3, res[5] = 0.375555
                Reader Thread No 4, res[1] = 0.0876856
Current state:   0.0000   0.0877   0.3115   0.6026   0.1202   0.3756   0.3524
                Reader Thread No 1, res[1] = 0.0876856
                Reader Thread No 4, res[4] = 0.120235
```

```
Next Session...
Work with 2 writer threads
Work with 3 reader threads
All threads make 5 reaps
Initial array:    0.0000   0.0000   0.0000   0.0000   0.0000
1       1       Really detached
        Writer Thread 1 is started
                Reader Thread 0 is started
                Reader Thread 1 is started
                Reader Thread 2 is started
        Writer Thread 0 is started
        Detached Thread is Started
Current state:   0.0000   0.0000   0.0000   0.0000   0.0000
                Reader Thread No 0, res[4] = 0
        Writer Thread No 0, res[4] = 0.909224
                Reader Thread No 1, res[3] = 0
        Writer Thread No 1, res[3] = 0.318815
Current state:   0.0000   0.0000   0.0000   0.3188   0.9092
```

```
                Reader Thread No 1, res[3] = 0.318815
                Reader Thread No 2, res[1] = 0.251343
Current state:   0.7132   0.2513   0.8286   0.3188   0.9092
                Reader Thread No 2, res[1] = 0.251343
                Reader Thread 0 is stopped
Current state:   0.7132   0.2513   0.8286   0.3188   0.9092
                Reader Thread No 2, res[3] = 0.318815
Current state:   0.7132   0.2513   0.8286   0.3188   0.9092
                Reader Thread No 1, res[0] = 0.713216
                Reader Thread 1 is stopped
Current state:   0.7132   0.2513   0.8286   0.3188   0.9092
Current state:   0.7132   0.2513   0.8286   0.3188   0.9092
                Reader Thread 2 is stopped
Result array :    0.7132   0.2513   0.8286   0.3188   0.9092
```