

Московский Авиационный Институт  
(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики  
Кафедра вычислительной математики и программирования

**Лабораторная работа №3**  
**по курсу «Операционные системы»**  
**III Семестр**

**Вариант 14**

Студент:	Короткевич Л. В.
Группа:	М80-208Б-19
Преподаватель:	Миронов Е.С
Оценка:	
Дата:	

# 1. Постановка задачи

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение потоков должно быть задано ключом запуска вашей программы.

Привести исследование зависимости ускорения и эффективности алгоритма от входящих данных и количества потоков. Получившиеся результаты объяснить.

## Вариант 14:

Есть колода из 52 карт, рассчитать экспериментально (метод Монте-Карло) вероятность того, что сверху лежат две одинаковых карты. Количество раундов подается с ключом.

# 2. Метод решения

*Используемые системные и библиотечные вызовы для выполнения работы:*

<code>int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);</code>	Функция <code>pthread_mutex_init()</code> инициализирует мьютекс с атрибутами, заданными <code>attr</code> . Если значение <code>attr</code> равно <code>NULL</code> , то используются атрибуты мьютекса по умолчанию
<code>int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);</code>	Функция <code>pthread_create()</code> запускает новый поток в вызывающем процессе. Новый поток начинает выполнение, вызывая <code>start_routine()</code> ; <code>arg</code> передается в качестве единственного аргумента <code>start_routine()</code> .
<code>int pthread_join(pthread_t thread, void **retval);</code>	Функция <code>pthread_join()</code> ожидает завершения указанного потока. Если этот поток уже завершен, то функция <code>pthread_join()</code> немедленно возвращается.

<pre>int pthread_mutex_destroy (pthread_mutex_t *mutex);</pre>	Функция pthread_mutex_destroy() уничтожает мьютекс; мьютекс становится, по сути, неинициализированным.
--	--

*Краткий алгоритм решения:*

- 1) Получение числа раундов, кол-ва потоков в кач-ве аргументов при запуске программы.
- 2) Инициализация мьютекса.
- 3) Создание заданного числа потоков.
  - 1) Моделирование перемешивания колоды карт в функции потоков.
  - 2) Заблокировать мьютекс.
  - 3) Инкрементировать результат, переменную count в случае успешного результата эксперимента.
  - 4) Разблокировать мьютекс.
- 4) Ожидание завершения потоков.
- 5) Уничтожение мьютекса.

*Несколько подробней об отдельной функции void \*thread\_func(void \*arg):*  
 Функция получает на вход единственный аргумент – число раундов. Создается массив целых чисел размера 52. Заполняется числами от 0 до 12 (каждого из числа в массиве будет ровно 4, сколько и мастей в стандартной колоде). Устанавливается в качестве базы для rand() текущее время. После чего нехитрым алгоритмом случайным образом перемешивается исходный массив. Далее мы сравниваем две крайние карты и, соответственно, увеличиваем в случае положительного рез-тата глобальную переменную count на 1, прежде заблокировав мьютекс во избежание “гонки”. Наконец, разблокируем мьютекс и по мере завершения необх. числа раундов завершим работу определенной нити.

*Вычисление теоретической вероятности для последующего сравнения точности полученных результатов при проведении экспериментов:*

На первое место ставится одна из 52 карт. Следом за ней - одна из 3 оставшихся с равным первой значением (масти полагаем незначащим параметром). Следом - 50 любых карт в случайном порядке. Итого:  $52 * 3 * 50!$  удовл. вариантов. Всего же –  $52!$ . Вероятность равняется  $52 * 3 * 50! / 52! = 3/51$  – приблизительно 5.9%.

### *Метрики параллельных вычислений:*

Метрики параллельных вычислений используется для оценки роста производительности, получаемого при параллельном решении задачи на  $p$  процессорах. Также они позволяют определить необходимое кол-во процессоров, используемых для решения конкретной задачи.

- $T_p$  – время выполнения на  $p$  различных вычислительных ядрах
- Ускорение:  $S_p = T_1/T_p$ ,  $S_p < p$
- Эффективность/загруженность:  $X_p = S_p/p$ ,  $X_p < 1$
- Закон Амдала (здесь ускорение показывает во сколько раз меньше времени потребуется параллельной прог-е для выполнения):

$$S_p = 1 / (g + (1 - g) / p)$$

$g$  – доля последовательных расчётов в программе,

$p$  – количество процессоров,

$S_p$  – производительность.

- Закон Густавсона — Барсиса (здесь укорение -- это увеличение объема выполненной задачи за постоянный промежуток времени)

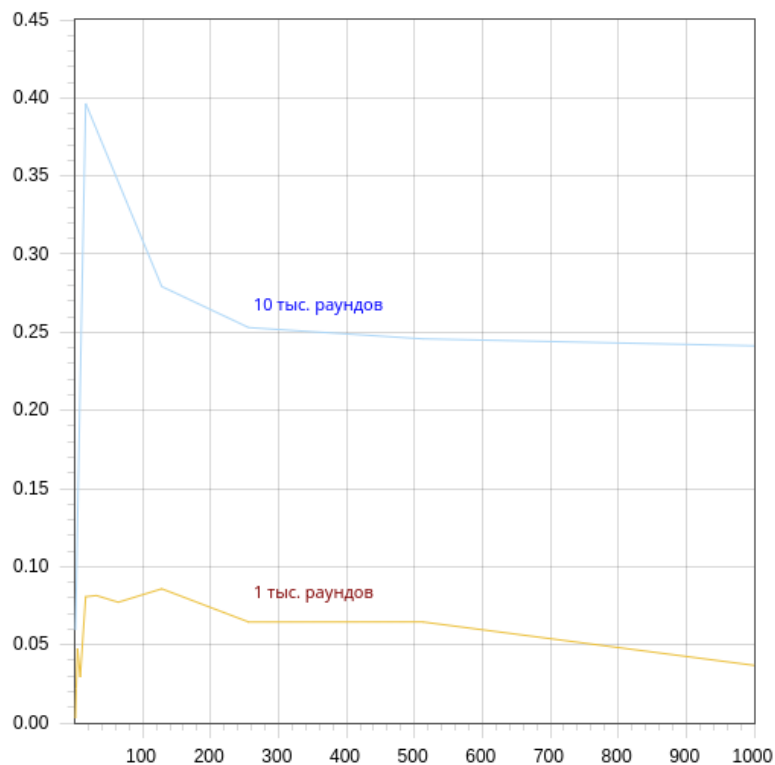
$$S_p = g + (1 - g)p = p + (1 - p)g$$

$g$  – доля последовательных расчётов в программе,

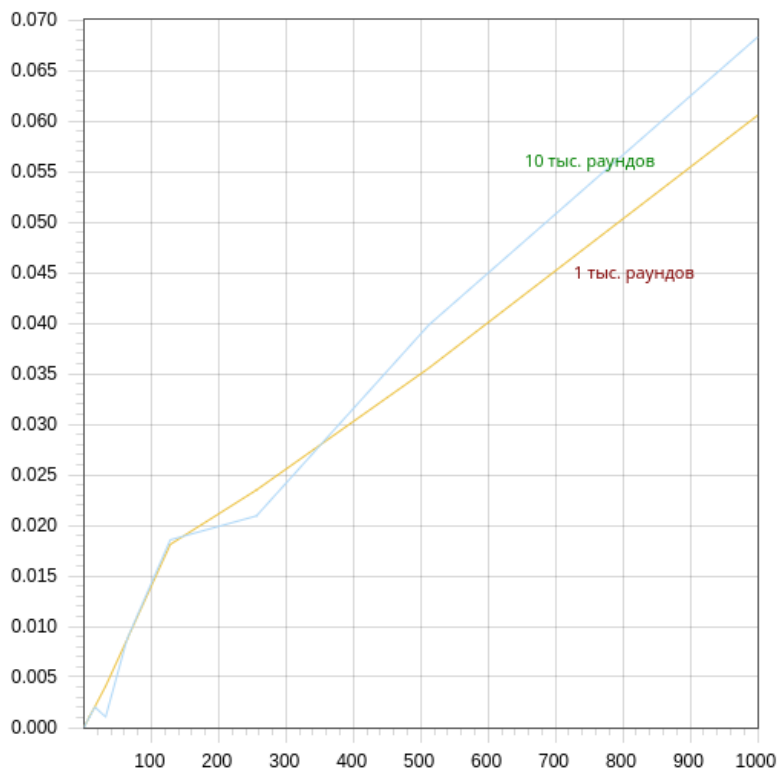
$p$  – количество процессоров.

### 3. Тестирование

График зависимости времени от числа потоков при фиксированном числе раундов:



Из интереса я написал программу того же содержания, но с использованием fork. Результаты ее тестирования приведены ниже.



Многопроцессорная версия проигрывает многопоточной программе. Часто многопроцессорные программы проще проектировать, если введены некоторые ограничения, однако они имеют тенденцию быть менее эффективными чем многопоточные их аналоги.

Теперь определим метрики параллельных вычислений на примере 1000 раундов.

<b>p</b>	<b>T1 (sec)</b>	<b>Тp (sec)</b>	Ускорение ( <b>Sp</b> =T1/Тp)	Эффективность ( <b>Xp</b> =Sp/p)
2	0.003267	0.021822	0,1497113005	0,07485565026
4	0.003267	0.047556	0,06869795609	0,01717448902
8	0.003267	0.029313	0,1114522567	0,01393153208
16	0.003267	0.08074	0,4046321526	0,02528950954
32	0.003267	0.081485	0,0400932687	0,001252914647
64	0.003267	0.077196	0,04232084564	0,0006612632131
128	0.003267	0.085800	0,03807692308	0,0002974759615
256	0.003267	0.064539	0,05062055501	0,000197736543
512	0.003267	0.063596	0,05137115542	0,0001003342879
1000	0.003267	0.036805	0,08876511344	0,00008876511344

Консоль:

```
/* using threads */
[leo@pc final]$ gcc -pthread -o main main.c
[leo@pc final]$ ./main 100 1
Probability equals to 0.050
Time spent: 0.001903 seconds
[leo@pc final]$ ./main 100 10
Probability equals to 0.080
Time spent: 0.008503 seconds
[leo@pc final]$ ./main 100 100
Probability equals to 0.080
Time spent: 0.015445 seconds
[leo@pc final]$ ./main 1000 1
Probability equals to 0.059
Time spent: 0.011920 seconds
[leo@pc final]$ ./main 1000 100
Probability equals to 0.064
Time spent: 0.015275 seconds
[leo@pc final]$ ./main 1000 1000
Probability equals to 0.046
Time spent: 0.095889 seconds
/* using forks */
[leo@pc using forks]$ ./main 1000 1
59/1000 experiments were successful
Probability equals to 5.90%
Time spent: 0.000147 seconds
[leo@pc using forks]$ ./main 1000 10
46/1000 experiments were successful
```

*Probability equals to 4.60%*  
*Time spent: 0.001292 seconds*  
*[leo@pc using forks]\$ ./main 1000 100*  
*39/1000 experiments were successful*  
*Probability equals to 3.90%*  
*Time spent: 0.012084 seconds*  
*[leo@pc using forks]\$ ./main 1000 1000*  
*25/1000 experiments were successful*  
*Probability equals to 2.50%*  
*Time spent: 0.068951 seconds*

## 4. Листинг программы

**main.c:**

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

const int DECK_SIZE = 52;

void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int count = 0;
pthread_mutex_t mutex;

void *thread_func(void *arg)
{
    int deck[DECK_SIZE];

    for (int k = 0; k < *((int *)arg); ++k)
    {
        for (int i = 0; i < DECK_SIZE; ++i)
        {
            deck[i] = (i + 1) % 13; // значения карт варьируются от 2 до 14 == от 0 до 12
        }

        srand(clock());

        for (int i = DECK_SIZE - 1; i >= 0; --i)
        {
            int j = (int)rand() % (i + 1);
            swap(&deck[i], &deck[j]);
        }

        if (deck[0] == deck[1])
        {
            pthread_mutex_lock(&mutex);
            ++count;
            pthread_mutex_unlock(&mutex);
        }
    }

    pthread_exit(0);
}

int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        perror("Usage: ./main <number of rounds> <number of threads>\n");
        exit(1);
    }

    clock_t begin = clock();

    pthread_t tid[atoi(argv[2])];

    if (pthread_mutex_init(&mutex, NULL) < 0)
```

```

{
    perror("Mutex init error");
    exit(1);
}

int rounds = atoi(argv[1]);
for (int i = 0; i < atoi(argv[2]); ++i)
{
    if (pthread_create(&tid[i], NULL, thread_func, &rounds) != 0)
    {
        perror("Can't create thread\n");
        exit(1);
    }
}
for (int i = 0; i < atoi(argv[2]); ++i)
{
    if (pthread_join(tid[i], NULL) != 0)
    {
        perror("Can't join threads");
        exit(1);
    }
}

if (pthread_mutex_destroy(&mutex) < 0)
{
    perror("Mutex destroy error");
    exit(1);
}

clock_t end = clock();
double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

printf("Probability equals to %.3lf\n", (double)count / (atoi(argv[1]) * atoi(argv[2])));
printf("Time spent: %lf seconds\n", time_spent);

return 0;
}

```

## multifork version:

```

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>

#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>

#define DECK_SIZE 52

int random_in_interval(int a, int b)
{
    return a + rand() % (b - a + 1);
}

void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        fprintf(stderr, "Usage: ./main <number of rounds> <number of threads>\n");
        exit(1);
    }

    int r = atoi(argv[1]) / atoi(argv[2]), n = atoi(argv[2]) + 1;

    int **fd = (int **)malloc(sizeof(int *) * n);
    for (int i = 0; i < n; ++i)
        fd[i] = (int *)malloc(sizeof(int) * 2);
    for (int i = 0; i < n; ++i)

```



```

{
    if (pipe(fd[i]) < 0)
    {
        perror("Pipe error");
        exit(1);
    }
}

clock_t begin = clock();
pid_t *pid = (pid_t *)malloc(sizeof(pid_t) * n);
for (int i = 1; i < n; ++i)
{
    pid[i] = fork();
    if (pid[i] < 0)
    {
        perror("Fork error");
        exit(1);
    }

    if (pid[i] == 0)
    {
        for (int j = 0; j < n; ++j)
        {
            if (j != i - 1)
            {
                close(fd[j][0]);
            }
            if (j != i)
            {
                close(fd[j][1]);
            }
        }

        int x;

        if (read(fd[i - 1][0], &x, sizeof(int)) < 0)
        {
            perror("Reading error");
            exit(1);
        }

        int deck[DECK_SIZE];
        for (int k = 0; k < r; ++k)
        {
            srand(clock());

            for (int m = 0; m < DECK_SIZE; ++m)
            {
                deck[m] = (m + 1) % 13;
            }

            for (int p = DECK_SIZE - 1; p >= 0; --p)
            {
                int s = (int)rand() % (p + 1);
                swap(&deck[s], &deck[p]);
            }

            if (deck[51] == deck[50])
            {
                x++;
            }
        }

        if (write(fd[i][1], &x, sizeof(int)) < 0)
        {
            perror("Writing error");
            exit(1);
        }

        close(fd[i - 1][0]);
        close(fd[i][1]);

        return 0;
    }
}

clock_t end = clock();
double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

for (int j = 0; j < n; ++j)

```

```

{
    if (j != 0)
    {
        close(fd[j][1]);
    }
    if (j != n - 1)
    {
        close(fd[j][0]);
    }
}

int x = 0;

if (write(fd[0][1], &x, sizeof(int)) < 0)
{
    perror("Writing origin x to child1 error");
    exit(1);
}
if (read(fd[n - 1][0], &x, sizeof(int)) < 0)
{
    perror("Reading processed x error");
    exit(1);
}

printf("%d/%d experiments were successful\n", x, r * (n - 1));
printf("Probability equals to %.2lf%%\n", 100.0 * x / r / (n - 1));
printf("Time spent: %lf seconds\n", time_spent);

close(fd[n - 1][0]);
close(fd[0][1]);

for (int i = 1; i < n; ++i)
{
    waitpid(pid[i], NULL, 0);
}

// memory free
free(pid);
for (int i = 0; i < n; ++i)
    free(fd[i]);
free(fd);

return 0;
}

```

## Вывод

По мере выполнения данной лабораторной работы я познал преимущества и недостатки многопоточности; освоил работу с мьютексами, их создание/удаление, использование во избежание некорректностей, нарушения последовательности выполнения задач потоками.

Во время тестирования осознал следующие вещи: несмотря на сильную погрешность в измерении эффективности работы программы из-за тонкостей работы “железа” (оперативная память – полная или пустая, процессор – занят чем-нибудь или нет), наблюдается ярко-выраженный рост скорости при увеличении кол-ва потоков, что нельзя сказать о многопроцессорной версии программы.

Логичным было наблюдать превосходство многопоточности над многопроцессорностью: несколько процессов запускаются с целью повышения отказоустойчивости приложения, а также с целью повышения безопасности, но не эффективности. Многопоточность же - вариант реализации вычислений, при котором для решения некоторой прикладной задачи запускаются и выполняются несколько независимых потоков вычислений, имеющих общую

память; причём выполнение происходит одновременно или псевдоодновременно, что дает выигрыш во времени.

Также потоки быстрее создаются и уничтожаются, чем процессы (в некоторых системах в 10-100 раз). Это особенно критично в задачах, требующих частое изменение количества потоков.

Потоки одного процесса имеют полный доступ к адресному пространству всего процесса, в том числе таблице дескрипторов, сигналы и даже к данным других потоков. Один поток, например, может напрямую записывать данные в стек другого потока. Этого никто не запрещает.

Однако, потоки имеют следующий недостаток: при параллельной работе нескольких потоков может произойти ситуация, когда потоки одновременно обращаются к одному ресурсу. И, если оба из них просто "читают", то вроде бы и ничего плохого в этом нет, но если один из них ведет запись, то у потоков может оказаться неактуальная информация, а следовательно, дальнейшая работа программы будет некорректной.

Для того, чтобы справиться с данной проблемой были придуманы примитивы синхронизации. В них входят критические области, барьеры, спин-блокировки, семафоры и т.д.

Наблюдательно, что для точной оценки вероятности (схожей с теоретической до сотых) достаточно всего одной тысячи раундов. Я был приятно удивлен тому, как просто проверять правдивость тех или иных теоретических утверждений с помощью компьютера. Полагаю, без многопоточности сей процесс занял бы большое время.

Интересно также было реализовывать метод Монте-Карло: полное моделирование ситуации, эксперимента для последующего получения результатов. Хотя в данной ЛР мне кажется избыточным полное моделирование процесса (можно было бы обойтись случайной генерацией двух крайних карт), в других случаях, как, например, вычисление числа «пи» с точностью до  $n$  знаков и других вычислительных задачах без этого никуда!