Московский Авиационный Институт
(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная информатика и программирование»
Факультет: «Информационные технологии и прикладная математика»

Лабораторная работа
Дисциплина: «Операционные системы»
3 семестр
Задание 3
Вариант 11

Группа:	М8О-208Б-18, №12
Студент:	Коростелев Дмитрий Васильевич
Преподаватель:	Миронов Евгений Сергеевич
Оценка:	
Дата:	29.11.2019

Москва, 2019

Содержание

1.Задание

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). При создании необходимо предусмотреть ключи, которые позволяли бы задать максимальное количество потоков, используемое программой. При возможности необходимо использовать максимальное количество возможных потоков. Ограничение потоков может быть задано или ключом запуска вашей программы, или алгоритмом.

Так же необходимо уметь продемонстрировать количество потоков, используемое вашей программой с помощью стандартных средств операционной системы.

Вариант 11 : На вход программе подаются игровое поле для игры "Крестики-нолики" и ход какого игрока сейчас идет. Программа должна выдать наиболее оптимальный ход для заданного игрока (если их несколько, то выдать все).

2.Адрес репозитория на GitHub

https://github.com/Dmitry4K/labOS3

3.Код программы

Source.c:

```
#define _CRT_SECURE_NO_WARNINGS

#include<stdio.h>
#include<stdlib.h>
#include<malloc.h>
#include<Windows.h>
#include<tchar.h>
#include<locale.h>
#include<time.h>
#include"GameTicTacToe.h"



int _tmain(int argc, _TCHAR* argv[]) {
	setlocale(LC_ALL, "rus");
	int** Field = NULL;
```

```
	Field = CreateField(Field);
	printf("Введите поле 3 x 3 :\n");
	printf("  v v v\n");

	for (int i = 0; i < 3; i++) {
		printf("> ");
		for (int j = 0; j < 3; j++)
			scanf("%d", &Field[i][j]);
	}

	printf("\nВведите игрока, который ходит следующим : ");
	int Player;
	scanf("%d", &Player);

	printf("Введите кол-во потоков доступное для рассчета : ");
	int ThreadCount;
	scanf("%d", &ThreadCount);

	HANDLE* Threads = malloc(sizeof(HANDLE)*ThreadCount);
	struct Step* Answers = NULL;
	int* AnswersSize = (int*)malloc(sizeof(int));
	*AnswersSize = 0;
	//TicTacToeBestStep(Field, Player, &ThreadCount, &Answers, AnswersSize);
	clock_t start = clock();
	ThreadFunctionForTicTacToeBestStep(Field, Player, &ThreadCount, &Answers,
AnswersSize);
	printf("\n");
	if (*AnswersSize == 0) {
		printf("Выигрышных ходов нет\n");
		ThreadFunctionForTicTacToeLoselessStep(Field, Player, &ThreadCount,
&Answers, AnswersSize);
		for (int i = 0; i < *AnswersSize; i++)
			printf("Беспроигрышный ход : %d %d\n", Answers[i].x, Answers[i].y);

	}
	else {
		for (int i = 0; i < *AnswersSize; i++)
			printf("Выигрышный ход : %d %d\n", Answers[i].x, Answers[i].y);
	}
	clock_t end = clock();
	printf("\nВремя затраченное на подсчет : %.0lf мс\n\n", (end - start) * 1000.0 /
(CLOCKS_PER_SEC));

	system("pause");
	return 0;
	 }
```

GameTicTacToe.h:

```
#pragma once
#include<Windows.h>

struct Step {
	int x, y;
};

struct ThParam {
	int **field;
	int player;
	int* count_threads;
	struct Step **answer_buffer;
	int *size_of_buf;
	HANDLE* threads;
	int* used_threads;
};
```

```
void ThreadFunctionForTicTacToeBestStep(int **field, int player, int* count_threads,
struct Step **answer_buffer, int* size_of_buf);
void ThreadFunctionForTicTacToeLoselessStep(int **field, int player, int* count_threads,
struct Step **answer_buffer, int* size_of_buf);
DWORD WINAPI TicTacToeLoselessStepThread(LPVOID LpParam);
void TicTacToeLoselessStep(LPVOID LpParam);
struct ThParam* CopyThreadParam(struct ThParam* Param);
DWORD WINAPI TicTacToeBestStepThread(LPVOID lpParam);
void TicTacToeBestStep(LPVOID lpParam);
//void TicTacToeBestStep(int **field, int player, int* count_threads, struct Step**
answer_buffer, int* size_of_buf);
int CheckClearPoint(int**);
int CheckWinner(int**);
void printField(int**);
int** DeleteField(int**);
int** CopyField(int**);
int** CreateField(int**);
```

GameTicTacToe.c

```
#include"GameTicTacToe.h"
#include<malloc.h>

CRITICAL_SECTION section = { 0 };
/*
struct Step* CalculateBestStep(int ** Field, int Player) {
	struct Step* res = NULL;
	for (int i = 0; i < 3; i++)
		for (int j = 0; j < 3; j++)
			if (Field[i][j] != 2 && Field[i][j] != 1) {
				Field[i][j] = Player;
				//printField(Field);
				//printf("%d\n", i);
				if (CheckWinner(Field) == Player) {
					//printf("winner: %d\n", Player);
					res = (struct Step*)malloc(sizeof(struct Step));
					res->x = i;
					res->y = j;
					return res;
				}
				int ** FieldC = CopyField(Field);
				res = CalculateBestStep(FieldC, (Player * 2) % 3);
				FieldC = DeleteField(FieldC);
				if (res == NULL) {
					res = (struct Step*)malloc(sizeof(struct Step));
					res->x = i;
					res->y = j;
```

```
                                return res;
                        }
                        Field[i][j] = 0;
                }
        return res;
}*/

void printField(int** Field) {
        for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 3; j++)
                        printf("%d ", Field[i][j]);
                printf("\n");
        }
}

int** DeleteField(int** Field) {
        for (int i = 0; i < 3; i++)
                free(Field[i]);
        free(Field);
        return NULL;
}

int** CopyField(int** Field) {
        int** res = (int**)malloc(sizeof(int*) * 3);
        for (int i = 0; i < 3; i++)
                res[i] = (int*)malloc(sizeof(int) * 3);
        for (int i = 0; i < 3; i++)
                for (int j = 0; j < 3; j++)
                        res[i][j] = Field[i][j];
        return res;
}

int CheckWinner(int ** Field) {
        for (int i = 0; i < 3; i++) {
                if (Field[i][0] == Field[i][1] && Field[i][0] == Field[i][2] && Field[i][0]
!= 0)
                        return Field[i][0];
                if (Field[0][i] == Field[1][i] && Field[0][i] == Field[2][i] && Field[0][i]
!= 0)
                        return Field[0][i];
        }
        if (Field[0][0] == Field[2][2] && Field[1][1] == Field[2][2] && Field[0][0] != 0)
                return Field[0][0];
```

```
	if (Field[1][1] == Field[0][2] && Field[1][1] == Field[2][0] && Field[1][1] != 0)
		return Field[1][1];
	return 0;
}


int** CreateField(int** Field) {
	Field = (int **)malloc(sizeof(int*) * 3);
	for (int i = 0; i < 3; i++)
		*(Field + i) = (int *)malloc(sizeof(int) * 3);
	return Field;
}

DWORD WINAPI TicTacToeBestStepThread(LPVOID LpParam) {
	struct ThParam *ThreadParam = (struct ThParam*)LpParam;
	for (int i = 0; i < 3; i++) {
		for (int j = 0; j < 3; j++) {
			if (ThreadParam->field[i][j] == 0) {
				int** c_field = CopyField(ThreadParam->field);
				c_field[i][j] = ThreadParam->player;
				//printField(c_field);
				if (CheckWinner(c_field) == ThreadParam->player) {
					if (*(ThreadParam->answer_buffer) != NULL)
						*(ThreadParam->answer_buffer) = (struct
Step*)realloc(*(ThreadParam->answer_buffer), (*(ThreadParam->size_of_buf) + 1) *
sizeof(struct Step));
					else
						*(ThreadParam->answer_buffer) = (struct
Step*)malloc(sizeof(struct Step));
					(*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].x = i;
					(*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].y = j;
					(*(ThreadParam->size_of_buf))++;
				}
				else {
					struct Step* answer_buffer_next = NULL;
					int* abf_size = (int*)malloc(sizeof(int));
					*abf_size = 0;
					struct ThParam * c_LpParam =
CopyThreadParam(ThreadParam);
					c_LpParam->field = c_field;
					c_LpParam->player = (ThreadParam->player * 2) % 3;
```

```
                                //c_LpParam->count_threads =
                                c_LpParam->answer_buffer = &answer_buffer_next;
                                c_LpParam->size_of_buf = abf_size;
                                int cur_thread = 0;
                                if (c_LpParam->used_threads < c_LpParam->count_threads)
{
                                        printf("%d ", *((*c_LpParam).used_threads));

                                        EnterCriticalSection(&section);
                                        *(c_LpParam->used_threads)     =     *(c_LpParam-
>used_threads) + 1;
                                        LeaveCriticalSection(&section);
                                        c_LpParam->threads[*(c_LpParam->used_threads)-1]
= CreateThread(NULL, 0, TicTacToeBestStepThread, c_LpParam, 0, NULL);

                                        cur_thread = *(c_LpParam->used_threads);
                                }
                                else {
                                        TicTacToeBestStep((LPVOID)c_LpParam);
                                }
                                if (cur_thread != 0)
                                        WaitForSingleObject(ThreadParam-
>threads[cur_thread - 1], INFINITE);
                                if (*abf_size == 0 && CheckClearPoint(c_field) > 0) {
                                        if (*(ThreadParam->answer_buffer) != NULL)
                                                *(ThreadParam->answer_buffer)            =
realloc(*(ThreadParam->answer_buffer), (*(ThreadParam->size_of_buf) + 1) * sizeof(struct
Step));
                                        else
                                                *(ThreadParam->answer_buffer)            =
malloc(sizeof(struct Step));
                                        (*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].x = i;
                                        (*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].y = j;
                                        (*(ThreadParam->size_of_buf))++;
                                }
                        }
                        c_field = DeleteField(c_field);
                }
        }
    }
    ExitThread(0);
```

```
}

void TicTacToeBestStep(LPVOID LpParam) {
	struct ThParam *ThreadParam = (struct ThParam*)LpParam;
	for (int i = 0; i < 3; i++) {
		for (int j = 0; j < 3; j++) {
			if (ThreadParam->field[i][j] == 0) {
				int** c_field = CopyField(ThreadParam->field);
				c_field[i][j] = ThreadParam->player;
				//printField(c_field);
				if (CheckWinner(c_field) == ThreadParam->player) {
					if (*(ThreadParam->answer_buffer) != NULL)
						*(ThreadParam->answer_buffer) = (struct
Step*)realloc(*(ThreadParam->answer_buffer), (*(ThreadParam->size_of_buf)+1) *
sizeof(struct Step));
					else
						*(ThreadParam->answer_buffer) = (struct
Step*)malloc(sizeof(struct Step));
					(*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].x = i;
					(*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].y = j;
					(*(ThreadParam->size_of_buf))++;
				}
				else {
					struct Step* answer_buffer_next = NULL;
					int* abf_size = (int*)malloc(sizeof(int));
					*abf_size = 0;
					struct ThParam * c_LpParam =
CopyThreadParam(ThreadParam);
					c_LpParam->field = c_field;
					c_LpParam->player = (ThreadParam->player * 2) % 3;
					//c_LpParam->count_threads =
					c_LpParam->answer_buffer = &answer_buffer_next;
					c_LpParam->size_of_buf = abf_size;
					TicTacToeBestStep((LPVOID)c_LpParam);
					if (*abf_size == 0 && CheckClearPoint(c_field) > 0) {
						if (*(ThreadParam->answer_buffer) != NULL)
							*(ThreadParam->answer_buffer) =
realloc(*(ThreadParam->answer_buffer), (*(ThreadParam->size_of_buf)+1) * sizeof(struct
Step));
						else
```

```
                                                *(ThreadParam->answer_buffer)            =
malloc(sizeof(struct Step));
                                        (*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].x = i;
                                        (*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].y = j;
                                        (*(ThreadParam->size_of_buf))++;
                                }
                        }
                        c_field = DeleteField(c_field);
                }
        }
    }
}


void ThreadFunctionForTicTacToeBestStep(int **field, int player, int* count_threads, struct
Step **answer_buffer, int* size_of_buf) {
    InitializeCriticalSection(&section);
    int*  used_threads = malloc(sizeof(int));
    *used_threads = 0;
    HANDLE* Threads = malloc(sizeof(HANDLE)*(*count_threads));
    struct ThParam*  ThreadParam = malloc(sizeof(struct ThParam));
    ThreadParam->field = field;
    ThreadParam->player = player;
    ThreadParam->count_threads = count_threads;
    ThreadParam->answer_buffer = answer_buffer;
    ThreadParam->size_of_buf = size_of_buf;
    ThreadParam->threads = Threads;
    ThreadParam->used_threads = used_threads;
    if (*count_threads > 0) {

        EnterCriticalSection(&section);
        *(ThreadParam->used_threads) = *(ThreadParam->used_threads) + 1;
        LeaveCriticalSection(&section);
        ThreadParam->threads[*(ThreadParam->used_threads) - 1] = CreateThread(NULL,
0, TicTacToeBestStepThread, ThreadParam, 0, NULL);

        WaitForSingleObject(Threads[0], INFINITE);
        DeleteCriticalSection(&section);
    }
    else {
        TicTacToeBestStep((LPVOID)ThreadParam);
```

```
	}
}
void ThreadFunctionForTicTacToeLoselessStep(int **field, int player, int* count_threads,
struct Step **answer_buffer, int* size_of_buf) {
	InitializeCriticalSection(&section);
	int*  used_threads = malloc(sizeof(int));
	*used_threads = 0;
	HANDLE* Threads = malloc(sizeof(HANDLE)*(*count_threads));
	struct ThParam*  ThreadParam = malloc(sizeof(struct ThParam));
	ThreadParam->field = field;
	ThreadParam->player = player;
	ThreadParam->count_threads = count_threads;
	ThreadParam->answer_buffer = answer_buffer;
	ThreadParam->size_of_buf = size_of_buf;
	ThreadParam->threads = Threads;
	ThreadParam->used_threads = used_threads;
	if (*count_threads > 0) {

		//EnterCriticalSection(&section);
		*(ThreadParam->used_threads) = *(ThreadParam->used_threads) + 1;
		//LeaveCriticalSection(&section);
		ThreadParam->threads[*(ThreadParam->used_threads) - 1] = CreateThread(NULL,
0, TicTacToeLoselessStepThread, ThreadParam, 0, NULL);

		WaitForSingleObject(Threads[0], INFINITE);
		DeleteCriticalSection(&section);
	}
	else {
		TicTacToeLoselessStep((LPVOID)ThreadParam);
	}
}


void TicTacToeLoselessStep(LPVOID LpParam) {
	struct ThParam *ThreadParam = (struct ThParam*)LpParam;
	for (int i = 0; i < 3; i++) {
		for (int j = 0; j < 3; j++) {
			if (ThreadParam->field[i][j] == 0) {
				int** c_field = CopyField(ThreadParam->field);
				c_field[i][j] = ThreadParam->player;
				//printField(c_field);
				if (CheckWinner(c_field) == ThreadParam->player) {
					if (*(ThreadParam->answer_buffer) != NULL)
```

```
                                        *(ThreadParam->answer_buffer)      =      (struct
Step*)realloc(*(ThreadParam->answer_buffer),    (*(ThreadParam->size_of_buf)    +    1)    *
sizeof(struct Step));
                                  else
                                        *(ThreadParam->answer_buffer)      =      (struct
Step*)malloc(sizeof(struct Step));
                                  (*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].x = i;
                                  (*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].y = j;
                                  (*(ThreadParam->size_of_buf))++;
                            }
                            else {
                                  struct Step* answer_buffer_next = NULL;
                                  int* abf_size = (int*)malloc(sizeof(int));
                                  *abf_size = 0;
                                  struct        ThParam        *        c_LpParam        =
CopyThreadParam(ThreadParam);
                                  c_LpParam->field = c_field;
                                  c_LpParam->player = (ThreadParam->player * 2) % 3;
                                  c_LpParam->answer_buffer = &answer_buffer_next;
                                  c_LpParam->size_of_buf = abf_size;
                                  TicTacToeLoselessStep((LPVOID)c_LpParam);
                                  if (*abf_size == 0) {
                                        if (*(ThreadParam->answer_buffer) != NULL)
                                              *(ThreadParam->answer_buffer)            =
realloc(*(ThreadParam->answer_buffer), (*(ThreadParam->size_of_buf) + 1) * sizeof(struct
Step));
                                        else
                                              *(ThreadParam->answer_buffer)            =
malloc(sizeof(struct Step));
                                        (*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].x = i;
                                        (*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].y = j;
                                        (*(ThreadParam->size_of_buf))++;
                                  }
                            }
                            c_field = DeleteField(c_field);
                      }
                }
          }
    }
}
```

```
DWORD WINAPI TicTacToeLoselessStepThread(LPVOID LpParam) {
	struct ThParam *ThreadParam = (struct ThParam*)LpParam;
	for (int i = 0; i < 3; i++) {
		for (int j = 0; j < 3; j++) {
			if (ThreadParam->field[i][j] == 0) {
				int** c_field = CopyField(ThreadParam->field);
				c_field[i][j] = ThreadParam->player;
				//printField(c_field);
				if (CheckWinner(c_field) == ThreadParam->player) {
					if (*(ThreadParam->answer_buffer) != NULL)
						*(ThreadParam->answer_buffer) = (struct
Step*)realloc(*(ThreadParam->answer_buffer), (*(ThreadParam->size_of_buf) + 1) *
sizeof(struct Step));
					else
						*(ThreadParam->answer_buffer) = (struct
Step*)malloc(sizeof(struct Step));
					(*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].x = i;
					(*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].y = j;
					(*(ThreadParam->size_of_buf))++;
				}
				else {
					struct Step* answer_buffer_next = NULL;
					int* abf_size = (int*)malloc(sizeof(int));
					*abf_size = 0;
					struct ThParam * c_LpParam =
CopyThreadParam(ThreadParam);
					c_LpParam->field = c_field;
					c_LpParam->player = (ThreadParam->player * 2) % 3;
					//c_LpParam->count_threads =
					c_LpParam->answer_buffer = &answer_buffer_next;
					c_LpParam->size_of_buf = abf_size;
					int cur_thread = 0;
					if (c_LpParam->used_threads < c_LpParam->count_threads)
{
						printf("%d ", *((*c_LpParam).used_threads));

						EnterCriticalSection(&section);
						*(c_LpParam->used_threads) = *(c_LpParam-
>used_threads) + 1;
```

```
                                        LeaveCriticalSection(&section);
                                        c_LpParam->threads[*(c_LpParam->used_threads)   -
1] = CreateThread(NULL, 0, TicTacToeBestStepThread, c_LpParam, 0, NULL);

                                        cur_thread = *(c_LpParam->used_threads);
                                }
                                else {
                                        TicTacToeLoselessStep((LPVOID)c_LpParam);
                                }
                                if (cur_thread != 0)
                                        WaitForSingleObject(ThreadParam-
>threads[cur_thread - 1], INFINITE);
                                if (*abf_size == 0) {
                                        if (*(ThreadParam->answer_buffer) != NULL)
                                                *(ThreadParam->answer_buffer)           =
realloc(*(ThreadParam->answer_buffer), (*(ThreadParam->size_of_buf) + 1) * sizeof(struct
Step));
                                        else
                                                *(ThreadParam->answer_buffer)           =
malloc(sizeof(struct Step));
                                        (*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].x = i;
                                        (*(ThreadParam->answer_buffer))[*(ThreadParam-
>size_of_buf)].y = j;
                                        (*(ThreadParam->size_of_buf))++;
                                }
                        }
                        c_field = DeleteField(c_field);
                }
        }
    }
    ExitThread(0);
}

/*
void  TicTacToeBestStep(int  **field,  int  player,  int*  count_threads,  struct  Step
**answer_buffer, int* size_of_buf) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (field[i][j] == 0) {
                int** c_field = CopyField(field);
                c_field[i][j] = player;
                //printField(c_field);
```

```
                        if (CheckWinner(c_field) == player) {
                                if (*answer_buffer != NULL)
                                        *answer_buffer             =             (struct
Step*)realloc(*answer_buffer, (*(size_of_buf) + 1)*sizeof(struct Step));
                                else
                                        *answer_buffer             =             (struct
Step*)malloc(sizeof(struct Step));
                                (*answer_buffer)[*(size_of_buf)].x = i;
                                (*answer_buffer)[*(size_of_buf)].y = j;
                                (*size_of_buf)++;
                        }
                        else {
                                struct Step* answer_buffer_next = NULL;
                                int* abf_size = (int*)malloc(sizeof(int));
                                *abf_size = 0;
                                TicTacToeBestStep(c_field,   (player   *   2)   %   3,
count_threads, &answer_buffer_next, abf_size);
                                if (*abf_size == 0 && CheckClearPoint(c_field) > 0) {
                                        if (*answer_buffer != NULL  )
                                                *answer_buffer  =  realloc(*answer_buffer,
(*(size_of_buf) + 1)*sizeof(struct Step));
                                        else
                                                *answer_buffer   =   malloc(sizeof(struct
Step));
                                        (*answer_buffer)[*(size_of_buf)].x = i;
                                        (*answer_buffer)[*(size_of_buf)].y = j;
                                        (*size_of_buf)++;
                                }
                        }
                        c_field = DeleteField(c_field);
                }
        }
    }
}*/

int CheckClearPoint(int ** field) {
    int res = 0;
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (field[i][j] == 0)
                res++;
    return res;
}
```

```
struct ThParam* CopyThreadParam(struct ThParam* Param) {
	struct ThParam* Res = malloc(sizeof(struct ThParam));
	Res->field = Param->field;
	Res->player = Param->player;
	Res->count_threads = Param->count_threads;
	Res->answer_buffer = Param->answer_buffer;
	Res->size_of_buf = Param->size_of_buf;
	Res->threads = Param->threads;
	Res->used_threads = Param->used_threads;
	return Res;
}
```

4.Результаты выполнения тестов

```
 1

 Введите поле 3 x 3 :
   v v v
 > 0 0 0
 > 0 0 0
 > 0 0 0

 Введите игрока, который ходит следующим : 1
 Введите кол-во потоков доступное для рассчета : 0

 Выигрышных ходов нет
 Беспроигрышный ход : 0 0
 Беспроигрышный ход : 0 1
 Беспроигрышный ход : 0 2
 Беспроигрышный ход : 1 0
 Беспроигрышный ход : 1 1
 Беспроигрышный ход : 1 2
 Беспроигрышный ход : 2 0
 Беспроигрышный ход : 2 1
 Беспроигрышный ход : 2 2

 Время затраченное на подсчет : 2525 мс

 Для продолжения нажмите любую клавишу . . .


 2

 Введите поле 3 x 3 :
   v v v
 > 0 0 0
 > 0 0 0
 > 0 0 0

 Введите игрока, который ходит следующим : 1
```

```
Введите кол-во потоков доступное для рассчета : 100000

Выигрышных ходов нет
Беспроигрышный ход : 0 0
Беспроигрышный ход : 0 1
Беспроигрышный ход : 0 2
Беспроигрышный ход : 1 0
Беспроигрышный ход : 1 1
Беспроигрышный ход : 1 2
Беспроигрышный ход : 2 0
Беспроигрышный ход : 2 1
Беспроигрышный ход : 2 2

Время затраченное на подсчет : 2357 мс

Для продолжения нажмите любую клавишу . . .

3

Введите поле 3 x 3 :
  v v v
> 1 0 2
> 0 0 2
> 0 1 0

Введите игрока, который ходит следующим : 1
Введите кол-во потоков доступное для рассчета : 0

Выигрышный ход : 2 2

Время затраченное на подсчет : 3 мс

Для продолжения нажмите любую клавишу . . .


4

Введите поле 3 x 3 :
  v v v
> 1 0 2
> 0 0 2
> 0 1 0

Введите игрока, который ходит следующим : 1
Введите кол-во потоков доступное для рассчета : 1000

Выигрышный ход : 2 2

Время затраченное на подсчет : 4 мс

Для продолжения нажмите любую клавишу . . .
```

5.Объяснение результатов работы программы

Программа получает на вход информацию о текующей ситуации на поле, далее вводится игрок, который ходит следующим и кол-во потоков, которое впослествие будет использовать программой, далее запускается сам алгоритм, который исходя из уже использованного кол-ва потоков либо создает новый поток, либо запускает фукнцию на главном потоке, каждый поток может создать еще 9 потоков, но сам «родительский поток» исходя из алгоритма, должен дожидаться завершения дочерних потоков, в связи с этим при небольших расчетах выгодней использовать меньшее кол-во потоков, так время на запуск потоков превышает время вычесления на одном потоке, а если посмотреть на тест, где поле не заполнено и нужно провести большое кол-во вычислений – выигрываем время производя вычисления сразу в несокльких потоках.

6.Вывод

Не сложно убедится, что умение пользования многопоточности могут, как ускорить, так и замедлить выполнение работы того или иного алгоритма. В связи с этим возникает понимание того, что нужно четко представлять, как использовать ту или иную технологию - уметь пользоваться ее средствами, применять в нужных ситуациях.