Отчет о выполнении лабораторной работы по изучению алгоритма Петерсона.

***V2***

Оглавление

[Введение 2](#_Toc181040814)

[Программа 3](#_Toc181040815)

[Аналитический подход к анализу конфликта 6](#_Toc181040816)

[Комбинаторный анализ 7](#_Toc181040817)

[Время исполнения и количество итераций 8](#_Toc181040818)

[Предложение по обходу конфликта 9](#_Toc181040819)

[Статистические исследования 11](#_Toc181040820)

[Заключение 12](#_Toc181040821)

[Литература 13](#_Toc181040822)

[Приложение А. 14](#_Toc181040823)

# Введение

**Алгоритм Петерсона** — это классический алгоритм для синхронизации процессов, который обеспечивает взаимное исключение для двух потоков. Он подходит для работы в средах с многозадачностью и предназначен для контроля доступа к общему ресурсу без использования стандартных средств синхронизации.

**Основные понятия:**

* **Взаимное исключение:** Гарантирует, что в данный момент времени только один поток может выполнять критическую секцию.
* **Локальные переменные:** Используются, чтобы указать, хочет ли поток войти в критическую секцию (флаг) и определить, чей черед заходить в критическую секцию (идентификатор потока).
* **Критическая секция:** Участок кода, который обращается к общему ресурсу.

Алгоритм:

1. Каждый поток устанавливает свой флаг в `true`, указывая, что он желает войти в критическую секцию.

2. Поток устанавливает значение переменной, указывающей, какой поток может входить в критическую секцию.

3. Поток проверяет, разрешен ли ему вход в критическую секцию. Если нет, он ждет.

4. Когда поток завершает работу в критической секции, он сбрасывает свой флаг.

# Программа

Работа программы (см. Листинг 1):

* Инициализация: Создается массив `want`, который использует индексы потоков для обозначения желания доступа к критической секции. Переменная `turn` указывает, чей сейчас ход.
* Функция потока (thread\_function): Каждый поток выполняет заданное количество итераций. В каждой итерации он пытается получить доступ к критической секции, выводит сообщение о входе и выходе из нее.
* Синхронизация: Используются массивы `want` и переменная `turn` для контроля доступа к критической секции.

Листинг 1 – Код программы

|  |
| --- |
| 1. #**include** <stdio.h> 2. #**include** <stdlib.h> 3. #**include** <pthread.h> 4. #**include** <unistd.h> 5. #**include** <time.h> 6. #**define** NUM\_THREADS 2 // Число потоков 7. #**define** NUM\_ITERATIONS 100000000 // Число итераций для каждого потока 8. #**define** WAIT\_TIME 1 // Время ожидания в секундах 9. int want[NUM\_THREADS] = {0, 0}; // Массив флагов для потоков 10. int turn; // Переменная, указывающая чей сейчас ход 11. **volatile** int conflict\_counter = 0; // Счетчик конфликтов 12. **volatile** int total\_counter = 0; 13. static int lim[ 16 ] = { 1, 13, 12, 9, 7, 15, 4, 3, 2, 11, 14, 5, 6, 8, 10, 0 }; 14. void \***thread\_function**(void \*arg) { 15. int id = \*(int \*)arg; 16. int n = 1231; 17. **for** (int i = 0; i < NUM\_ITERATIONS; ++i) { 18. **for** (**volatile** int j = 10\*lim[n & 15] + 1; j >= 0; --j) { 19. } 20. want[id] = 1; // Я хочу войти 21. turn = 1 - id; // Теперь очередь другого потока 22. **while** (want[1 - id] && turn == 1 - id); // Ждем, пока другой поток не закончится 23. n = \_\_atomic\_fetch\_add( &total\_counter, 1, \_\_ATOMIC\_SEQ\_CST ); 24. // Критическая секция 25. // printf("Поток %d вошёл в критическую секцию\n", id); 26. // sleep(WAIT\_TIME); // Имитация работы в критической секции 27. // printf("Поток %d покинул критическую секцию\n", id); 28. **for** (**volatile** int j = 10\*lim[n & 15] + 1; j >= 0; --j) { 29. } 30. // Проверка на конфликты 31. **if** ( 1 == ( 1 & n ) ) { 32. \_\_atomic\_fetch\_add( &conflict\_counter, 1, \_\_ATOMIC\_SEQ\_CST ); 33. // printf("Конфликт обнаружен между потоками!\n"); 34. } 35. \_\_atomic\_fetch\_add( &total\_counter, 1, \_\_ATOMIC\_SEQ\_CST ); 36. // Завершение ввода 37. want[id] = 0; // Я вышел 38. } 39. **return** NULL; 40. } 41. int **main**() { 42. pthread\_t threads[NUM\_THREADS]; 43. int thread\_ids[NUM\_THREADS] = {0, 1}; 44. // Запуск потоков 45. **for** (int i = 0; i < NUM\_THREADS; ++i) { 46. pthread\_create(&threads[i], NULL, thread\_function, (void \*)&thread\_ids[i]); 47. } 48. // Ожидание завершения потоков 49. **for** (int i = 0; i < NUM\_THREADS; ++i) { 50. pthread\_join(threads[i], NULL); 51. } 52. printf("Всего конфликтов: %d\n", conflict\_counter); 53. **return** 0; 54. } |

# Аналитический подход к анализу конфликта

В коде используется алгоритм Петерсона для управления доступом потоков к критической секции. Конфликты могут возникать из-за неправильного порядка доступа к ресурсам, что приводит к гонке данных.

Рассмотрим основные этапы конфликта:

* Желание войти в критическую секцию: Каждый поток устанавливает свой флаг want[id] = 1, чтобы указать, что он хочет войти в критическую секцию.
* Установка очереди: После этого поток устанавливает turn для другого потока, что может привести к ситуации, когда оба потока пытаются войти в критическую секцию почти одновременно.
* Ожидание: Оба потока могут входить в цикл ожидания параллельно, пока другой поток не завершит выполнение в критической секции. Это может привести к "взаимной блокировке", где оба потока ждут друг друга.
* Конфликты: Конфликт в коде регистрируется, когда 1 == (1 & n). Это происходит, если оба потока одновременно достигают критической секции, не успев корректно обработать флаги.

# Комбинаторный анализ

При большом количестве итераций, шансы на возникновение конфликта увеличиваются. Предположим, что оба потока выполняются параллельно и случайным образом пытаются войти в критическую секцию. В таком случае вероятность одновременного запроса на вход резко возрастает.

Используем простую модель:

* Каждую итерацию один поток может войти в критическую секцию с вероятностью p.
* Вероятность того, что оба потока попытаются войти одновременно может быть оценена как

Можно заметить, что чем больше итераций, тем выше вероятность конфликта следует исследовать опытных потоков.

# Время исполнения и количество итераций

Чтобы измерить время выполнения, можно использовать функции вроде `clock()` или `gettimeofday()`, чтобы зафиксировать начало и окончание выполнения потоков. Эти данные можно использовать для перестройки логики и оценки времени выполнения критической секции (см. Листинг 2):

Листинг 2 – время исполнения и количество итераций

|  |
| --- |
| 1. #**include** <sys/time.h> 2. **struct** **timeval** **start**, **end**; 3. // Перед запуском потоков 4. gettimeofday(&start, NULL); 5. // После завершения потоков 6. gettimeofday(&end, NULL); 7. long seconds = end.tv\_sec - start.tv\_sec; 8. long micros = end.tv\_usec - start.tv\_usec; 9. long elapsed = seconds \* 1000000 + micros; // В микросекундах 10. printf("Время выполнения: %ld микро-секунд\n", elapsed); |

# Предложение по обходу конфликта

В общем случае видим:

* Оптимизация флагов: Вместо использования флагов, можно применять атомарные операции или семафоры для синхронизации доступа.
* Уменьшение времени в критической секции: Минимизация время, которое потоки проводят в критической секции, чтобы уменьшить вероятность конфликта.
* Увеличение числа потоков: Рапределение нагрузки между большим количеством потоков может уменьшить нагрузку на критическую секцию и снизить вероятность конфликтов.

Есть несколько стратегий, которые можно реализовать для улучшения синхронизации и уменьшения конфликтов:

* Использование мьютексов: Вместо использования простых флагов, использовать мьютексы. Это более надежный способ управления доступом к критическим секциям и может устранить гонки состояний.  
    
  Листинг 3 – Пример использование mutex

|  |
| --- |
| 1. pthread\_mutex\_t mutex; 2. pthread\_mutex\_init(&mutex, NULL); 3. ... 4. // Вход в критическую секцию 5. pthread\_mutex\_lock(&mutex); 6. // Критическая секция 7. pthread\_mutex\_unlock(&mutex); |

* Установка ожидания: Вместо постоянного опроса (busy-waiting) можно установить механизм ожидания, который будет блокироваться до тех пор, пока не освободится критическая секция.
* Использование условных переменных: Это позволяет потокам ожидать, пока другая часть системы не выполнит определенное условие, что минимизирует конфликты.

# Статистические исследования

Для проведения статистических исследований поведения был выбран способ отметки количества итераций вложенного цикла на каждой итерации.

Листинг 4 – Сбор статистических данных

// Ожидание, пока не пройдет очередь  
while (want[1 - id] && turn == 1 - id) {  
 // Увеличиваем счетчик операций во время ожидания  
 if (id == 0) {  
 operation\_cnt\_id\_zero++;  
 } else {  
 operation\_cnt\_id\_one++;  
 }  
}

Далее в качестве тестовых данных были выбраны следующие значения числа итераций {100, 200, 500, 1000, 2000, 5000, 10000, 20000, 40000, 80000, 160000, 350000, 1000000, 5000000};

Исследование показало (см. Рисунок 1, 2, 3)

Изображение выглядит как текст, линия, График, снимок экрана

Автоматически созданное описание

Изображение выглядит как линия, текст, График, диаграмма

Автоматически созданное описание

Изображение выглядит как текст, линия, График, диаграмма

Автоматически созданное описание

# Заключение

Таким образом, ключевыми шагами, направленными на минимизацию конфликтов в многопоточном окружении, являются внедрение более высокоуровневых методов синхронизации, таких как мьютексы и атомарные операции, а также активное использование анализа времени выполнения и частоты конфликтов.

Проведенный анализ, включая количественные данные о производительности, показывает, что предложенные меры могут значительно увеличить эффективность и надежность многопоточных приложений. Например, данные, полученные в результате эмпирического эксперимента, иллюстрируют связь между ростом объема операций и возникновением конфликтов, что подчеркивает важность грамотного управления многопоточностью.

Так, наблюдаемые показатели для различных объемов операций демонстрируют увеличение числа конфликтов при увеличении нагрузки, что особенно заметно при более крупных масштабах обработки данных. Например, для 1 миллиона операций зафиксировано 45 конфликтов, а при 5 миллионах операций количество конфликтов достигло 291. Эти данные подтверждают необходимость реализации более эффективных алгоритмов управления многопоточностью, что имеет критическое значение в условиях постоянно растущей сложности программного обеспечения.

Данная лабораторная работа подчеркивает актуальность дальнейших исследований в области многопоточности, ориентированных на создание оптимизированных алгоритмов и методов, которые обеспечат высокую производительность и минимизацию конфликтов, что в свою очередь может существенно повысить устойчивость и безопасность программных систем в условиях многопоточной работы.

Такой подход придаст вашим выводам научный характер и продемонстрирует глубокое понимание исследуемых вопросов.

# Литература

<https://cyberleninka.ru/article/n/analiz-mehanizmov-organizatsii-parallelnyh-vychisleniy/viewer>

<https://habr.com/ru/companies/intel/articles/583286/>

<https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D1%8B_%D0%B2%D0%B7%D0%B0%D0%B8%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B8%D1%81%D0%BA%D0%BB%D1%8E%D1%87%D0%B5%D0%BD%D0%B8%D1%8F>

<https://www.tstu.ru/book/elib3/mm/2016/evdokimov/site/page39.39.html>

<https://studfile.net/preview/880820/page:34/>

# Приложение А.

import numpy as np  
import matplotlib.pyplot as plt  
  
# Открываем файл и читаем данные  
data = np.loadtxt('results.txt')  
  
# Извлекаем столбцы  
x = data[:, 0] # Первый столбец: количество итераций  
y1 = data[:, 1] # Второй столбец: количество операций для потока 0  
y2 = data[:, 2] # Третий столбец: количество операций для потока 1  
  
# Вычисляем общее количество операций  
total\_operations = y1 + y2  
  
# Вычисляем относительные значения  
relative\_y1 = (y1 / total\_operations) \* 100 # Процент операций для потока 0  
relative\_y2 = (y2 / total\_operations) \* 100 # Процент операций для потока 1  
  
# Строим график для абсолютных значений  
plt.figure(figsize=(14, 6))  
  
# Абсолютные значения  
plt.subplot(1, 2, 1)  
plt.plot(x, y1, marker='o', label='Поток 0', color='b')  
plt.plot(x, y2, marker='x', label='Поток 1', color='r')  
plt.title('Количество операций для потоков 0 и 1 в зависимости от числа итераций')  
plt.xlabel('Количество итераций')  
plt.ylabel('Количество операций')  
plt.legend()  
plt.grid(True)  
  
# Относительные значения  
plt.subplot(1, 2, 2)  
plt.plot(x, relative\_y1, marker='o', label='Поток 0', color='b')  
plt.plot(x, relative\_y2, marker='x', label='Поток 1', color='r')  
plt.title('Относительное количество операций для потоков 0 и 1 в зависимости от числа итераций')  
plt.xlabel('Количество итераций')  
plt.ylabel('Относительное количество операций (%)')  
plt.legend()  
plt.grid(True)  
  
# Сохраняем график в формате PNG  
plt.savefig('plot.png')  
  
# Показываем график  
plt.show()

Листинг А – Отрисовка графиков

# Приложение Б.

#include <stdio.h>  
#include <stdlib.h>  
#include <pthread.h>  
#include <unistd.h>  
  
#define NUM\_THREADS 2 // Количество потоков  
#define WAIT\_TIME 1 // Время ожидания в "операциях"  
#define MAX\_TESTS 14 // Максимальное количество тестов  
#define MAX\_ITERATIONS 2500 // Максимальное количество итераций для теста  
  
int want[NUM\_THREADS] = {0, 0}; // Массив флагов для потоков  
int turn; // Переменная, указывающая чей сейчас ход  
volatile int conflict\_counter = 0; // Счетчик конфликтов  
volatile int total\_counter = 0;  
volatile int operation\_cnt\_id\_zero = 0; // Счетчик операций для потока 0  
volatile int operation\_cnt\_id\_one = 0; // Счетчик операций для потока 1  
  
void \*thread\_function(void \*arg) {  
 int id = \*(int \*)arg;  
  
 int iterations = \*(int \*)(arg + sizeof(int));  
  
 for (int i = 0; i < iterations; ++i) {  
 // Выполнение операций  
 for (volatile int j = 0; j < WAIT\_TIME; ++j) {  
 // Пустой цикл ожидания  
 }  
  
 want[id] = 1; // Я хочу войти  
 turn = 1 - id; // Теперь очередь другого потока  
  
 // Ожидание, пока не пройдет очередь  
 while (want[1 - id] && turn == 1 - id) {  
 // Увеличиваем счетчик операций во время ожидания  
 if (id == 0) {  
 operation\_cnt\_id\_zero++;  
 } else {  
 operation\_cnt\_id\_one++;  
 }  
 }  
  
 // Критическая секция  
 conflict\_counter++;  
 // Параллельные операции  
 total\_counter++;  
  
 want[id] = 0; // Я выходил  
 }  
  
 return NULL;  
}  
  
void run\_tests\_1() {  
 FILE \*file = fopen("results.txt", "w");  
 int iterations\_arr[MAX\_TESTS] = {100, 200, 500, 1000, 2000, 5000, 10000, 20000, 40000, 80000, 160000, 350000, 1000000, 5000000};  
  
 for (int t = 0; t < MAX\_TESTS; t++) {  
 int iterations = iterations\_arr[t];  
 pthread\_t threads[NUM\_THREADS];  
 operation\_cnt\_id\_zero = 0;  
 operation\_cnt\_id\_one = 0;  
 conflict\_counter = 0;  
 total\_counter = 0;  
  
 // Передаем количество итераций в аргументах потокам  
 int thread\_args[NUM\_THREADS][2] = {{0, iterations}, {1, iterations}};  
  
 for (int i = 0; i < NUM\_THREADS; i++) {  
 pthread\_create(&threads[i], NULL, thread\_function, (void\*)thread\_args[i]);  
 }  
  
 for (int i = 0; i < NUM\_THREADS; i++) {  
 pthread\_join(threads[i], NULL);  
 }  
  
 fprintf(file, "%d %d %d\n", iterations, operation\_cnt\_id\_zero, operation\_cnt\_id\_one);  
 }  
  
 fclose(file);  
}  
  
void run\_tests\_2() {  
 FILE \*file = fopen("results.txt", "w");  
 int iterations\_arr[MAX\_TESTS] = {5000000, 5000000, 5000000, 5000000, 5000000, 5000000, 5000000, 5000000, 5000000, 5000000, 5000000, 5000000, 5000000, 5000000};  
  
 for (int t = 0; t < MAX\_TESTS; t++) {  
 int iterations = iterations\_arr[t];  
 pthread\_t threads[NUM\_THREADS];  
 operation\_cnt\_id\_zero = 0;  
 operation\_cnt\_id\_one = 0;  
 conflict\_counter = 0;  
 total\_counter = 0;  
  
 // Передаем количество итераций в аргументах потокам  
 int thread\_args[NUM\_THREADS][2] = {{0, iterations}, {1, iterations}};  
  
 for (int i = 0; i < NUM\_THREADS; i++) {  
 pthread\_create(&threads[i], NULL, thread\_function, (void\*)thread\_args[i]);  
 }  
  
 for (int i = 0; i < NUM\_THREADS; i++) {  
 pthread\_join(threads[i], NULL);  
 }  
  
 fprintf(file, "%d %d %d\n", iterations, operation\_cnt\_id\_zero, operation\_cnt\_id\_one);  
 }  
  
 fclose(file);  
}  
  
void plot\_results() {  
 FILE \*pipe = popen("gnuplot -persistent", "w");  
 if (pipe == NULL) {  
 fprintf(stderr, "Не удалось открыть поток для gnuplot\n");  
 return;  
 }  
  
 fprintf(pipe, "set title 'Количество операций ожидания'\n");  
 fprintf(pipe, "set xlabel 'Количество итераций'\n");  
 fprintf(pipe, "set ylabel 'Количество операций'\n");  
 fprintf(pipe, "set style data linespoints\n");  
 fprintf(pipe, "plot 'results.txt' using 1:2 title 'Поток 0', '' using 1:3 title 'Поток 1'\n");  
  
 pclose(pipe);  
}  
  
int main() {  
 run\_tests\_1();  
// run\_tests\_2();  
// plot\_results();  
  
 return 0;  
}

Листинг Б – Сбор статистики