Отчет о выполнении лабораторной работы по изучению алгоритма Петерсона.

Оглавление

[Введение 2](#_Toc179829872)

[Программа 3](#_Toc179829873)

[Аналитический анализ конфликта 5](#_Toc179829874)

[Комбинаторный анализ 5](#_Toc179829875)

[Время исполнения и количество итераций 6](#_Toc179829876)

[Предложение по обходу конфликта 7](#_Toc179829877)

[Заключение 8](#_Toc179829878)

# 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 анализ 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 simultanously, 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, подумайте о том, чтобы 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.

# Заключение

Таким образом, решающие шаги по минимизации конфликтов в многопоточном окружении включают внедрение более высокоуровневых методов синхронизации, таких как мьютексы и атомарные операции, а также активное использование анализа времени выполнения и частоты конфликтов.

Проведенный анализ и предложенные меры позволяют значительно улучшить производительность и надёжность многопоточных приложений, что имеет важное значение в условиях возрастающей сложности программного обеспечения.

Данная лабораторная работа подчеркивает необходимость дальнейших исследований в этой области, направленных на разработку более эффективных алгоритмов управления многопоточностью.

# 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/>