Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу
«Операционные системы»

Студент: Велиев Рауф Рамиз оглы
Группа: М8О-209Б-23
Вариант: 4
Преподаватель: Миронов Евгений Сергеевич
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2024

Содержание

Репозиторий

https://github.com/velievrauf/OS/tree/main/lab_2

Постановка задачи

Цель работы

Приобретение практических навыков в:

· Управление потоками в ОС

· Обеспечение синхронизации между потоками

Задание

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработке использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение потоков должно быть задано ключом запуска программы.
Необходимо уметь продемонстрировать количество потоков, используемых программой, с помощью стандартных средств операционной системы.
Привести исследование зависимости ускорения и эффективности алгоритма от входныхх данных и количества потоков. Объяснить получившиеся результаты.
Отсортировать массив чисел при помощи TimSort.

Общий метод и алгоритм решения

Для сортировки использован алгоритм TimSort, который комбинирует сортировку вставками и слияние блоков. Массив делится на части, которые сортируются параллельно с использованием потоков. После сортировки блоков, они сливаются в один отсортированный массив. Количество потоков регулируется пользователем, что позволяет ускорить сортировку для больших данных.

Исходный код

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

#define RUN 32

int compare(const void *a, const void *b) {
   return (*(int*)a - *(int*)b);
}

void insertionSort(int arr[], int left, int right) {
   for (int i = left + 1; i <= right; i++) {
      int key = arr[i];
      int j = i - 1;
      while (j >= left && arr[j] > key) {
         arr[j + 1] = arr[j];
         j--;
      }
      arr[j + 1] = key;
   }
}

void merge(int arr[], int left, int mid, int right) {
   int len1 = mid - left + 1, len2 = right - mid;
   int *leftArr = (int *)malloc(len1 * sizeof(int));
   int *rightArr = (int *)malloc(len2 * sizeof(int));

   for (int i = 0; i < len1; i++)
      leftArr[i] = arr[left + i];
   for (int i = 0; i < len2; i++)
      rightArr[i] = arr[mid + 1 + i];

   int i = 0, j = 0, k = left;
   while (i < len1 && j < len2) {
      if (leftArr[i] <= rightArr[j]) {
```

```
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

    while (i < len1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }

    while (j < len2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }

    free(leftArr);
    free(rightArr);
}

void timSort(int arr[], int n) {
    for (int i = 0; i < n; i += RUN)
        insertionSort(arr, i, (i + RUN - 1) < (n - 1) ? (i + RUN - 1) : (n - 1));

    for (int size = RUN; size < n; size = 2 * size) {
        for (int left = 0; left < n; left += 2 * size) {
            int mid = left + size - 1;
            int right = (left + 2 * size - 1) < (n - 1) ? (left + 2 * size - 1) : (n - 1);

            if (mid < right)
                merge(arr, left, mid, right);
        }
```

```
    }
}

typedef struct {
    int *arr;
    int start;
    int end;
} ThreadData;

void *sortArray(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    timSort(data->arr + data->start, data->end - data->start);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    if (argc != 3) {
        printf("Usage: %s <array size> <number of threads>\n", argv[0]);
        return 1;
    }

    int arr_size = atoi(argv[1]);
    int max_threads = atoi(argv[2]);

    int *arr = (int *)malloc(arr_size * sizeof(int));
    if (!arr) {
        printf("Memory allocation failed\n");
        return 1;
    }
    for (int i = 0; i < arr_size; i++) {
        arr[i] = rand() % 1000;
    }

    pthread_t threads[max_threads];
    ThreadData threadData[max_threads];
    int segment_size = arr_size / max_threads;
```

```
    clock_t start_time = clock();

    for (int i = 0; i < max_threads; i++) {
        threadData[i].arr = arr;
        threadData[i].start = i * segment_size;
        threadData[i].end = (i == max_threads - 1) ? arr_size : (i + 1) * segment_size;

        pthread_create(&threads[i], NULL, sortArray, (void *)&threadData[i]);
    }

    for (int i = 0; i < max_threads; i++) {
        pthread_join(threads[i], NULL);
    }

    timSort(arr, arr_size);

    clock_t end_time = clock();
    double time_taken = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    printf("Time taken for sorting with %d threads: %.5f seconds\n", max_threads, time_taken);

    printf("Sorted Array: \n");
    for (int i = 0; i < arr_size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr);
    return 0;
}
```

Демонстрация работы программы

./lab2 10 2

Time taken for sorting with 2 threads: 0.00088 seconds

Sorted Array:

335 383 386 421 492 649 777 793 886 915

./lab2 10 4

Time taken for sorting with 4 threads: 0.00175 seconds

Sorted Array:

335 383 386 421 492 649 777 793 886 915

./lab2 20 4

Time taken for sorting with 4 threads: 0.00172 seconds

Sorted Array:

27 59 172 335 362 383 386 421 426 492 540 649 690 736 763 777 793 886 915 926

./lab2 20 8

Time taken for sorting with 8 threads: 0.00350 seconds

Sorted Array:

27 59 172 335 362 383 386 421 426 492 540 649 690 736 763 777 793 886 915 926

Вывод

В этой лабораторной работе был использован язык Си для реализации многозадачности с помощью потоков для сортировки массива. Применение потоков позволило нам разделить задачу на несколько частей.

В процессе работы была реализована параллельная сортировка с использованием алгоритма TimSort. При увеличении числа потоков, в некоторых случаях время выполнения увеличивалось из-за накладных расходов на создание и управление потоками. Это показало, что для получения реального ускорения важно правильно подбирать количество потоков в зависимости от объема данных и мощности системы.