Лабораторная работа № 3
по курсу "Операционные системы":

Выполнил студент группы 08-208 МАИ *Куликов Алексей*.

Цель работы

Приобретение практических навыков в:

1. Управление потоками в ОС
2. Обеспечение синхронизации между потоками

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.

В качестве конкретного варианта задания предлагается создание программы, сортирующей массив строк при помощи четно-нечетной сортировки Бетчера (вариант 5).

Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

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

Описание программы

Прооограмма осуществляет ввод слов в импровизированный вектор до конца файла. Затем запускается четно-нечетная сортировака Бэтчера. После этого отсортированные слова выводятся в стандартный выходной поток.

Сначала программа проверяет опции, переданные ей в качестве аргументов. Если среди ключей присутсвует ключ `-t`, то следующее за ним значение принимается как ограничение количества потоков при выполнении сортировки.

Затем, пока не достигнут конец файла, считываются слова и записываются в вектор. Вектор увеличивает свою емкость по мере надобности.

Далее незадействованные элементы в векторе устанавливаются в `NULL`. Это необходимо для корректной работы сортировки Бэтчера т.к. данная реализация работает на количестве данных равном степеням двойки.

Далее вызывется сама сортировка Бэтчера для объема данных равного следующей за реальным количеством данных степени двойки. Например для 56 вызывается сортировка для 64 строк, из которых последние 8 проинициализированы `NULL`'ами.

Сама сортировка слиянием Бэтчера аналогична простой сортировки слиянием за исключением самой процедуры слиянием.

Слияние Бэтчера – сама по себе рекурсивная процедура. Суть его в том, что есть две отсортированных последовательности, и нужно их слить в одну полностью отсортированную. Для этого при помощи обратной перетасовки сведем процедуру слияния к вдвое меньшей задаче. И так до тех пор, пока размер сливаемых кусочков не станет 2. Тогда можно выполнить процедуру сравнения-замены. А далее, на обратном ходе рекурсии осуществляем перетасовку. Таким образом на каждом шаге после перетасовки получаем почти отсортированную последовательность. Остается только применить процедуру сравнения-замены для всех пар элементов с индексами $(i, i+1)$, $i = 1, 3, ...n-1$. Т.е. для каждой внутренней пары в массиве.

Далее отсортированный массив выводится в стандартный выходной поток. Занятые ресурсы освобождаются.

Листинг

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <math.h>
#include <string.h>

#define THREADS_LIMIT   8
#define STR_SIZE        128

int string_compare_nullable(char *str1, char *str2);
int next_2pow(int n);
void shuffle(char **arr, int l, int r);
void unshuffle(char **arr, int l, int r);
void comp_exch_str(char **a, char **b);
void batcher_merge(char **arr, int l, int m, int r);
void* merge_sort_parallel(void *args);
void merge_sort(char **arr, int l, int r);

int thread_count = 0;
pthread_mutex_t thread_count_locker;
int threads_limit = THREADS_LIMIT;

int main(int argc, char const *argv[])
{
    if(argc == 3){
        if(strcmp(argv[1], "-t") == 0){
            threads_limit = atoi(argv[2]);
        }
    }
    int status = 0;
    status = pthread_mutex_init(&thread_count_locker, NULL);
    if(status){
        fprintf(stderr, "ERROR: mutex initialization\n");
        return EXIT_FAILURE;
    }
```

```
    int capacity = 1;
    int size = 0;
    char **arr = (char **)malloc(sizeof(char *) * capacity);
    if(!arr){
        fprintf(stderr, "ERROR: bad allocation\n");
        return EXIT_FAILURE;
    }
    while (1)
    {
        arr[size] = (char *)malloc(sizeof(char) * (STR_SIZE + 1));
        if(!arr[size]){
            fprintf(stderr, "ERROR: bad allocation\n");
            return EXIT_FAILURE;
        }
        if (scanf("%s", arr[size]) != 1){
            free(arr[size]);
            break;
        }
        ++size;
        if (size == capacity){
            capacity *= 2;
            arr = (char **)realloc(arr, sizeof(char *) * capacity);
            if (!arr)
            {
                fprintf(stderr, "ERROR: bad allocation\n");
                return EXIT_FAILURE;
            }
        }
    }
    for (int i = size; i < capacity; ++i)
        arr[i] = NULL;

    merge_sort(arr, 0, (int)pow(2, next_2pow(size)) - 1);

    for (int i = 0; i < size; ++i)
        printf("%s ", arr[i]);
    printf("\n");
    for(int i = 0; i < size; ++i){
        free(arr[i]);
    }
    free(arr);
    pthread_mutex_destroy(&thread_count_locker);

    return 0;
}

int string_compare_nullable(char *str1, char *str2){
    if(str1 == NULL)
        return 1;
    else if(str2 == NULL)
        return -1;
    else
        return strcmp(str1, str2);
}

int next_2pow(int n)
{
    double l = log2(n);
    if (l - (double)(int)(l) == 0.0)
        return (int)l;
    else
        return (int)l + 1;
}

void shuffle(char **arr, int l, int r)
```

```
{
    int count = r - l + 1;
    int m = (r + l) / 2;
    char **temp = (char **)malloc(count * sizeof(char*));
    for (int i = 0, k = 0; i + l <= m; ++i, k += 2)
    {
        temp[k] = arr[l + i];
        temp[k + 1] = arr[m + i + 1];
    }
    for (int i = 0; i < count; ++i)
        arr[l + i] = temp[i];
    free(temp);
}

void unshuffle(char **arr, int l, int r)
{
    int count = r - l + 1;
    int m = count / 2;
    char **temp = (char **)malloc(count * sizeof(char*));
    for (int i = 0, k = l; k < r; ++i, k += 2)
    {
        temp[i] = arr[k];
        temp[i + m] = arr[k + 1];
    }
    for (int i = 0; i < count; ++i)
        arr[l + i] = temp[i];
    free(temp);
}

void comp_exch_str(char **a, char **b)
{
    if (string_compare_nullable(*a, *b) > 0)
    {
        char* temp = *a;
        *a = *b;
        *b = temp;
    }
}

void batcher_merge(char **arr, int l, int m, int r)
{
    if (l + 1 == r)
    {
        comp_exch_str(&arr[l], &arr[r]);
    }
    if (l + 2 > r)
        return;
    unshuffle(arr, l, r);
    batcher_merge(arr, l, (l + m) / 2, m);
    batcher_merge(arr, m + 1, (m + r + 1) / 2, r);
    shuffle(arr, l, r);
    for (int i = l + 1; i < r; i += 2)
    {
        comp_exch_str(&arr[i], &arr[i + 1]);
    }
}

typedef struct
{
    char **arr;
    int l, r;
} sort_data;

void *merge_sort_parallel(void *args)
{
```

```
    sort_data *data = (sort_data *)args;
    merge_sort(data->arr, data->l, data->r);
    return NULL;
}

void merge_sort(char **arr, int l, int r){
    int m = (r + l) / 2;
    if (l == r)
        return;

    fprintf(stderr, "thread count: %d\n", thread_count);

    int is_threaded = 0;
    pthread_t thread_left;
    pthread_t thread_right;
    int thread_available = 0;
    int status = 0;
    pthread_mutex_lock(&thread_count_locker);
    if(thread_count < threads_limit){
        thread_available = threads_limit - thread_count;
        if(thread_available >= 2)
            thread_count += 2;
        else if(thread_available == 1)
            ++thread_count;
    }
    pthread_mutex_unlock(&thread_count_locker);

    if (thread_count > threads_limit)
    {
        fprintf(stderr, "ERROR: thread limit exceeded: %d\n", thread_count);
        return;
    }
    if (thread_available >= 2)
    {
        is_threaded = 1;
        sort_data data_left = {arr, l, m};
        sort_data data_right = {arr, m + 1, r};

        status = pthread_create(&thread_left, NULL, merge_sort_parallel, &data_left);
        if(status){
            fprintf(stderr, "ERROR: thread creation failed\n");
            return;
        }
        status = pthread_create(&thread_right, NULL, merge_sort_parallel, &data_right);
        if (status)
        {
            fprintf(stderr, "ERROR: thread creation failed\n");
            return;
        }
    }
    else if(thread_available == 1){
        is_threaded = 1;
        sort_data data_left = {arr, l, m};
        status = pthread_create(&thread_left, NULL, merge_sort_parallel, &data_left);
        if (status)
        {
            fprintf(stderr, "ERROR: thread creation failed\n");
            return;
        }
        merge_sort(arr, m + 1, r);
    }
    else{
        merge_sort(arr, l, m);
        merge_sort(arr, m + 1, r);
    }
```

```
    if (is_threaded){
        if(thread_available >= 2){
            status = pthread_join(thread_left, NULL);
            if (status)
            {
                fprintf(stderr, "ERROR: thread joining failed\n");
                return;
            }
            status = pthread_join(thread_right, NULL);
            if (status)
            {
                fprintf(stderr, "ERROR: thread joining failed\n");
                return;
            }
        }else if(thread_available == 1){
            status = pthread_join(thread_left, NULL);
            if (status)
            {
                fprintf(stderr, "ERROR: thread joining failed\n");
                return;
            }
        }
        pthread_mutex_lock(&thread_count_locker);
        if(thread_available >= 2)
            thread_count -= 2;
        else if(thread_available == 1)
            thread_count--;
        pthread_mutex_unlock(&thread_count_locker);
    }

    batcher_merge(arr, l, m, r);
}
```

Демонстрация работы

```
alex@alex:~/temp/OS/lab3$ cat inp
bad boys bad boy what you gonna do
alex@alex:~/temp/OS/lab3$ ./main < inp -t 8
bad bad boy boys do gonna what you
```

Strace

```
19098 clone(child_stack=0x7f4edeeeafb0, flags=CLONE_VM|CLONE_FS|CLONE_FILES|
CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|
CLONE_CHILD_CLEARTID, parent_tidptr=0x7f4edeeeb9d0, tls=0x7f4edeeeb700,
child_tidptr=0x7f4edeeeb9d0) = 19100
19098 clone(child_stack=0x7f4ede6e9fb0, flags=CLONE_VM|CLONE_FS|CLONE_FILES|
CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|
CLONE_CHILD_CLEARTID, parent_tidptr=0x7f4ede6ea9d0, tls=0x7f4ede6ea700,
child_tidptr=0x7f4ede6ea9d0) = 19101
19100 clone(child_stack=0x7f4eddee8fb0, flags=CLONE_VM|CLONE_FS|CLONE_FILES|
CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|
CLONE_CHILD_CLEARTID, parent_tidptr=0x7f4eddee99d0, tls=0x7f4eddee9700,
child_tidptr=0x7f4eddee99d0) = 19102
19100 clone(child_stack=0x7f4edb0e5fb0, flags=CLONE_VM|CLONE_FS|CLONE_FILES|
```

```
CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|
CLONE_CHILD_CLEARTID, parent_tidptr=0x7f4edb0e69d0, tls=0x7f4edb0e6700,
child_tidptr=0x7f4edb0e69d0) = 19103
19103 +++ exited with 0 +++
19102 +++ exited with 0 +++
...
```

Выводы

Основная причина использования потоков состоит в том, что многим программам необходимо совершать какие-либо действия параллельно, при этом не исключаются периодические блокировки части из них, возникающие по самым разным причинам, будь то ожидание ввода, запрос к диску и т.п. При этом программа не должна «зависать», в это время можно же занять ее чем-то другим.

Такую же возможность предоставляют процессы, но потоки обладают рядом преимуществ подобных задачах.

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 обеспечивают (псевдо-)параллелные операции.

Основной прирост производительности достигается в задачах, предполагающих большое количество операций ввода/вывода либо объемные вычислительные задачи. В таких случаях программа не будет последовательно выполняться шаг за шагом, напротив, несколько шагов будут выполняться одновременно. Особенно это заметно на многоядерных системах, в которых вычисления действительно могут происходить одновременно.

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.

Так же потоки имеют следующий недостаток: при параллельной работе нескольких потоков может произойти ситауция, 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 программмы будет некорректной.

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

Все выше перечисленные операции для работы с потоками и примитивы синхронизации реализованы в библиотеке `pthread`.

Основные компоненты библиотеки, необходимые для работы с потоками:

1. `pthread_create` – создание потока;

2. `pthread_join` – соединение потоков (поток, откуда вызывна функция дожидается указанного);

3. `pthread_mutex_lock` – блокирование мьютекса;

4. `pthread_mutex_unlock` – блокирование мьютекса.

и т.д.

Таких средств управления межпоточного взаимодействия довольно много и со всеми из них познакомиться не довелось. Надеюсь деведется в будущем.

Итак, в ходе работы над данной задачей, были получены знания и практические навыки использовании и управлении потоками в UNIX системе, а так же начальные навыки распараллеливания вычислений.