УДК 004.6 DOI: 10.31799/2077-5687-2024-5-60-66

ПРЕДЛОЖЕНИЯ ПО МОДИФИКАЦИИ СИСТЕМЫ ПРОСМОТРА И АНАЛИЗА ДИАГНОСТИЧЕСКОЙ ИНФОРМАЦИИ НА ОСНОВЕ ПРИМЕНЕНИЯ ГИБРИДНОЙ БАЗЫ ДАННЫХ

Н. Е. Шахомирова, А. В. Шахомиров
Санкт-Петербургский государственный университет аэрокосмического приборостроения

В работе рассмотрена возможность разработки гибридной системы хранения данных средства просмотра и анализа диагностической информации экспериментального программного компонента «Система автоматизированного проектирования бортовых вычислительных сетей космических аппаратов» автоматизированного комплекса проектирования и моделирования бортовых космических сетей SpaceWire (САПР SANDS). Показано, что предварительное преобразование данных из существующего представления в виде текстовых файлов в двоичное представление в виде таблиц и массивов данных, возможно организовать в виде многопоточного приложения, которое сначала преобразует все файлы диагностической информации в свой собственный формат гибридной базы данных, а затем позволит использовать эту информацию частями, на основе запросов к базе данных.

Ключевые слова: гибридные базы данных, многопоточные приложения, синхронизация.

Для цитирования:
Шахомирова, Н. Е. Предложения по модификации системы просмотра и анализа диагностической информации на основе применения гибридной базы данных / Н. Е. Шахомирова, А. В. Шахомирова // Системный анализ и логистика. – 2024. – № 5(43). – с. 60-66. DOI: 10.31799/2077-5687-2024-5-60-66.

PROPOSALS FOR MODIFICATION OF VIEWING AND ANALYZING DIAGNOSTIC INFORMATION SYSTEM, BASED ON THE USAGE OF A HYBRID DATABASE

N. E. Shakhomirova, A. V. Shakhomirov
St. Petersburg State University of Aerospace Instrumentation

The paper considers the possibility of developing a hybrid data storage system for viewing and analyzing diagnostic information of the experimental software component "Computer-aided design system for on-board computer networks of spacecraft" of the automated design and modeling complex for on-board SpaceWire space networks (SANDS CAD). It is shown that the preliminary transformation of data from an existing representation in the form of text files into a binary representation in the form of tables and data arrays can be organized in the form of a multithreaded application that first converts all diagnostic information files into its own hybrid database format, and then allows using this information in parts, based on database queries.

Keywords: hybrid databases, multithreaded applications, synchronization.

For citation:
Shakhomirova, N. E. Proposals for modification of viewing and analyzing diagnostic information system, based on the usage of a hybrid database / N. E. Shakhomirova, A. V. Shakhomirov // System analysis and logistics. – 2024. – № 5(43). – p. 60-66. DOI: 10.31799/2077-5687-2024-5-60-66.

Введение

Общая цель проекта – проектирование и разработка средства просмотра диагностический информации (в виде графического приложения с применением гибридной системы управления базами данных) для анализа диагностической информации экспериментального программного компонента «Система автоматизированного проектирования бортовых вычислительных сетей космических аппаратов» автоматизированного комплекса проектирования и моделирования бортовых космических сетей SpaceWire (САПР SANDS) [1].

Цели и задачи настоящей работы. На данном этапе проектирования и разработки требуется проанализировать возможность реализации основного функционала системы анализа диагностической информации САПР SANDS (загрузку данных в оперативную память из внешнего текстового файла, являющегося результатом работы САПР, и сохранение в файл в двоичном виде) на основе гибридного представления [2, 3], то есть использовать

высокоскоростные in-memory tables [4] и долговременное файловое хранение. Анализ текущего состояния системы просмотра диагностической информации показывает необходимость замены статического формата отчётов диагностической информации на динамический формат, который предполагает создание страницы отчёта во время её запроса пользователем. Это позволит выводить для анализа результатов моделирования только конкретные запрошенные данные, а не весь объём диагностической информации САПР. Кроме того, такой подход позволит иметь временную статистику последовательных запусков моделирования САПР SANDS, а не только один последний расчет.

Поток – это независимый путь выполнения внутри процесса. В рамках одного процесса может одновременно выполняться несколько потоков. Процесс использует все свои системные ресурсы (такие как адресное пространство памяти, код и глобальные данные) совместно со всеми этими потоками. Поток выполняет процедуру асинхронно по отношению к другим потокам, выполняющим те же или другие процедуры в рамках процесса.[5] Каждый поток имеет свои собственные специфические характеристики и атрибуты, такие как идентификатор потока, приоритет планирования, значение номера ошибки, привязки к конкретному потоку и системные ресурсы, необходимые для поддержки этого единого потока управления. Поток является базовой единицей, которой система выделяет процессорное время. Планирование выполнения потоков основано на приоритете потоков и доступности процессора, а в многопроцессорной системе выполнение разных потоков может быть запланировано на каждом процессоре.[5]

Многопоточность (multithreading) – это способность операционной системы или программного приложения выполнять несколько потоков (независимых задач) одновременно. Каждый поток способен исполнять собственный код и операции, что способствует эффективному использованию ресурсов центрального процессора и повышению общей производительности системы.[6]

Мьютексы (mutual exclusion objects) – это механизмы синхронизации, применяемые в многопоточном программировании для предотвращения одновременного доступа нескольких потоков к одному ресурсу. Мьютекс позволяет только одному потоку получить доступ к ресурсу, в то время как другие потоки ожидают освобождения этого ресурса.[6]

Программа реализует многопоточную систему для чтения данных из текстового файла, их временного хранения в буфере и записи в базу данных. Программа состоит из трех файлов: «main.c», «database.h» и «database.c». Основное внимание уделяется организации данных, алгоритмам многопоточности и структурам данных, используемым для хранения и управления информацией. В текущий момент реализация выполнена для операционных систем семейства *nix, с применением библиотеки pthread, для управления потоками, мьютексами, условными переменными и синхронизацией между потоками. В дальнейшем планируется адаптация программного кода под операционную систему семейства Windows.

Основные компоненты

1. Файл «main.c» выполняет функции инициализации, создания потоков для чтения и записи данных, а также измерения времени выполнения операций.
2. Заголовочный файл «database.h» содержит определения структур данных и объявления функций для работы с базой данных.
3. Файл «database.c» реализует функции для работы с базой данных, включая создание таблиц, вставку данных, выборку данных, сохранение и освобождение ресурсов.

Структуры данных в проекте

Структура для хранения информации о столбце таблицы, включающая имя столбца и тип данных.

```
typedef struct {
      char name[MAX_NAME_LEN];
```

```
        char type[MAX_NAME_LEN];
    } Column;
```

Предполагается использовать массив «columns» для определения столбцов таблиц, например:

```
    columns[] = {
        {"index", "INTEGER"},
        {"value", "TEXT"},
        {"timestamp", "INTEGER"}
    };
```

Структура для хранения строк данных. Используется для организации данных в страницах с фиксированным размером (4096 байт). Каждая страница содержит массив строк данных и счетчик количества строк:

```
    typedef struct {
        char **data;
        int row_count;
    } Page;
```

Структура для хранения информации о таблице, включая имя таблицы, количество столбцов, строк и страниц, а также массив структур Column и Page:

```
    #define MAX_COLUMNS 10
    typedef struct {
        char name[MAX_NAME_LEN];
        int column_count;
        int row_count;
        int page_count;
        Column columns[MAX_COLUMNS];
        Page *pages;
    } Table;
```

Главная структура для базы данных, содержащая массив таблиц и счетчик количества таблиц. Максимальное количество таблиц определяется значением MAX_TABLES.

```
    #define MAX_TABLES 50
    typedef struct {
        int table_count;
        Table tables[MAX_TABLES];
    } Database;
```

Буферизация данных

Для временного хранения данных предполагается использовать кольцевой буфер фиксированного размера (8192 байта). Буферизация реализована с помощью массива «buffer» и указателей на начало «buffer_start» и на конец буфера «buffer_end».

```
    #define BUFFER_SIZE 8192
    char buffer[BUFFER_SIZE];
```

Многопоточность

Многопоточность [6] позволяет программе выполнять несколько задач одновременно, увеличивая производительность и эффективность работы. В нашем случае будем использовать два потока:

- Поток чтения «read_file_thread». Этот поток читает данные из текстового файла «example.simlog» и записывает их в буфер. Цикл работы потока следующий: 1) Открывает файл для чтения. 2) Читает строки из файла. 3) Записывает прочитанные строки в буфер. 4) Сигнализирует об окончании данных в буфере.
- Поток записи «write_to_db_thread». Этот поток читает данные из буфера и записывает их в таблицы базы данных. Поток выполняет следующие действия: 1) Извлекает данные из буфера. 2) Разделяет данные на строки длиной 8 символов. 3)

Вставляет данные в таблицу «T_example_simlog», добавляя при этом индекс и временную метку.

```
//поток чтения файла
void *read_file_thread(void *arg) {
    FILE *file = fopen(FILENAME, "r");
    if (!file) {
        perror("Failed to open file");
        return NULL;
    }
    char bit_buffer[9];
    while (fgets(bit_buffer, sizeof(bit_buffer), file)) {
        bit_buffer[strcspn(bit_buffer, "\n")]='\0'; //null termination
        write_to_buffer(bit_buffer, strlen(bit_buffer));
    }
    fclose(file);
    write_to_buffer(NULL, -1); // Signal end of file
    return NULL;
}

//поток записи в таблицы в «оперативной памяти»
void *write_to_db_thread(void *arg) {
    int index = 0;
    while (1) {
        pthread_mutex_lock(&buffer_mutex);
        while (buffer_start == buffer_end && !buffer_full && !buffer_done) {
            pthread_cond_wait(&buffer_not_empty, &buffer_mutex);
        }
        if (buffer_done && buffer_start == buffer_end && !buffer_full) {
            pthread_mutex_unlock(&buffer_mutex);
            break; // Exit the loop if done and buffer is empty
        }
        Table *table = &db.tables[0];
        char row[9];  // To hold 8 bits plus null terminator
        memset(row, 0, sizeof(row)); // Initialize the array with nulls
        int row_index = 0;
        while ((buffer_start!=buffer_end||buffer_full)&&row_index<8) {
            row[row_index++] = buffer[buffer_start];
            buffer_start = (buffer_start + 1) % BUFFER_SIZE;
            if (buffer_full && buffer_start == buffer_end) {
                buffer_full = 0;
                pthread_cond_signal(&buffer_not_full);
            }
        }
        row[row_index] = '\0';
        pthread_mutex_unlock(&buffer_mutex);
        // Process data and insert into the database
        if (row_index > 0) {
            char *values[3];
            values[0] = (char *)malloc(20); // Index
            snprintf(values[0], 20, "%d", index++);
            values[1] = strdup(row);        // Save the bit string
            values[2] = (char *)malloc(20); // Timestamp
            snprintf(values[2], 20, "%lld", (long long)time(NULL));
            if (insert_into_table(table->name, values) != 0) {
                fprintf(stderr, "Failed to insert row into table\n");
            }
            for (int i = 0; i < 3; i++) {
                free(values[i]);
            }
        }
    }
```

```
    return NULL;
}
```

Синхронизация потоков

Синхронизация между потоками происходит с применением мьютексов и условных переменных. Это гарантирует корректную работу и управление общими ресурсами, и предотвращает конфликты доступа к данным. В многопоточных приложениях потоки могут одновременно обращаться к общим ресурсам, таким как переменные, объекты или файлы, что создает риск ошибок, если несколько потоков пытаются одновременно изменить или прочитать один и тот же ресурс. Основная сложность заключается в непредсказуемости порядка выполнения операций. Потоки могут «пересекаться» в своих действиях, что приводит к состояниям гонки и некорректным результатам. Поэтому важно правильно синхронизировать доступ к общим ресурсам.[6]

Для предотвращения состояния гонки (когда несколько потоков одновременно обращаются к одному и тому же ресурсу, при этом хотя бы один поток изменяет его значение), применяются механизмы синхронизации, такие как мьютексы [6].

Мьютексы и условные переменные для синхронизации потоков: `pthread_mutex_t` и `pthread_cond_t`. Создание потоков: `pthread_create`. Ожидание завершения потоков: `pthread_join`.

```
void write_to_buffer(const char *data, int length) {
    pthread_mutex_lock(&buffer_mutex);
    if (data == NULL && length == -1) { //Special case: signal the end of data
        buffer_done = 1;
        pthread_cond_signal(&buffer_not_empty);
        pthread_mutex_unlock(&buffer_mutex);
        return;
    }
    for (int i = 0; i < length; ++i) {
        buffer[buffer_end] = data[i];
        buffer_end = (buffer_end + 1) % BUFFER_SIZE;
        if (buffer_end == buffer_start) {
            buffer_full = 1;
        }
    }
    pthread_cond_signal(&buffer_not_empty);
    pthread_mutex_unlock(&buffer_mutex);
}
```

Страничная организация памяти

Страничная организация памяти применяется для оптимизации управления данными в базе данных. В рассматриваемом случае таблицы структурированы в страничное представление фиксированного размера, который на текущий момент составляет 4096 байт.

Страничная организация памяти для таблиц позволяет:

- снизить фрагментацию памяти;
- ускорить доступ к данным благодаря снижению количества операций ввода-вывода;
- упростить управление памятью при операциях добавления и удаления данных.

```
#define PAGE_SIZE 4096
void add_page(Table *table) {
    table->pages=realloc(table->pages,(table->page_count+1)*sizeof(Page));
    if (!table->pages) {
        perror("Failed to allocate memory for pages");
        exit(1);
    }
    table->pages[table->page_count].data=malloc(PAGE_SIZE*sizeof(char*));
```

```
    if (!table->pages[table->page_count].data) {
        perror("Failed to allocate memory for page data");
        exit(1);
    }
    table->pages[table->page_count].row_count=0;
    table->page_count++;
}
```

Последовательный алгоритм работы

1. Инициализация базы данных
   Вызов функции init_database() для инициализации структуры базы данных. Определение структуры столбцов таблицы и создание таблицы «T_example_simlog» с помощью функции create_table().
2. Запуск потоков
   Создание и запуск потока чтения данных из файла «read_file_thread». Создание и запуск потока записи данных в базу данных «write_to_db_thread».
3. Чтение данных из файла
   Открытие файла «example.simlog» для чтения. Чтение строк из файла и запись их в буфер с помощью функции write_to_buffe().
4. Запись данных в базу данных
   Извлечение данных из буфера и разбор на строки длиной 8 символов. Вставка данных в таблицу «T_example_simlog» с добавлением индекса и временной метки.
5. Измерение времени выполнения
   Измерение времени чтения данных из файла и записи в буфер. Измерение времени записи данных в базу данных и выполнения выборки данных.
6. Сохранение базы данных
   Сохранение состояния базы данных в бинарный файл «db.bin» с помощью функции «save_database()»
7. Освобождение ресурсов
   Освобождение всех выделенных ресурсов и памяти с помощью функции «free_database()».

Заключение

В ходе работы рассмотрен один из вариантов трансформации средства просмотра и анализа диагностической информации экспериментального программного компонента «Система автоматизированного проектирования бортовых вычислительных сетей космических аппаратов» автоматизированного комплекса проектирования и моделирования бортовых космических сетей SpaceWire (САПР SANDS) от статического представления в виде текстовых файлов до динамического представление в виде таблиц и массивов данных, которые реализуются средствами разрабатываемой гибридной базы данных. В качестве одной из возможностей по увеличению скорости обработки и поиска информации в гибридной базе данных рассматривается внедрение системы индексирования таблиц на основе современных алгоритмов индексирования [7]. Индексирование таблиц планируется к внедрению в следующей версии программы. Кроме того, планируются проведение исследования влияния размерности разделяемых страниц памяти на общую производительность системы при выполнении операций выборки данных.

СПИСОК ЛИТЕРАТУРЫ

1. *Оленев В. Л.* Подходы к развитию систем автоматизированного проектирования и моделирования бортовых сетей / В. Л. Оленев, А. В. Шахомиров // Системный анализ и логистика. – №4(30). – СПб.: ГУАП., 2021 – с. 76–86. DOI: 10.31799/2077-5687-2021-4-76-86.

2. Software testing help. What Is Hybrid Database? List of The Best Hybrid Databases. [Электронный ресурс] – URL: https://www.softwaretestinghelp.com/hybrid-database/ (дата обращения: 04.12.2024)

3. *Мухин А. М.* Технология структурирования и обработки транскриптомных данных на основе гибридного использования RDBMS и NoSQL подходов / А. М. Мухин, М. А. Генаев, Д. А. Рассказов, С. А. Лашин, Д. А. Афонников // Математическая биология и биоинформатика. – 2020. – Т. 15, № 2. – С. 455–470. DOI: 10.17537/2020.15.455

4. Microsoft Learn Challenge. In-memory database systems and technologies. [Электронный ресурс] – URL: https://learn.microsoft.com/en-us/sql/relational-databases/in-memory-database (дата обращения: 04.12.2024)

5. *Funkenhauser M.* Pthread Support in Microsoft Windows Services for UNIX Version 3.5 [Электронный ресурс] // Microsoft Learn Challenge. 2007. 12 May. – URL: https://learn.microsoft.com/en-us/previous-versions/tn-archive/bb463209(v=technet.10) (дата обращения 04.12.2024)

6. *Стручков М.* Введение в многопоточность: Преимущества, проблемы и ключевые концепции [Электронный ресурс] // Блог по программированию на Java. 2024. 9 октября. – URL: https://struchkov.dev/blog/ru/introduction-to-multithreading/ (дата обращения: 04.12.2024)

7. *Birler A., Fent P., Schmidt T., Neumann T.* Simple, Efficient, and Robust Hash Tables for Join Processing // DaMoN '24: Proceedings of the 20th International Workshop on Data Management on New Hardware. – Santiago, 2024. – № 4. – P. 1-9. DOI: 10.1145/3662010.3663442

ИНФОРМАЦИЯ ОБ АВТОРАХ

Шахомирова Наталья Евгеньевна
Ассистент кафедры аэрокосмических компьютерных и программных систем
Санкт-Петербургский государственный университет аэрокосмического приборостроения
Россия, 190000, Санкт-Петербург, ул. Большая Морская, д.67, лит. А
E-mail: n.shah2024@yandex.ru

Шахомиров Андрей Викторович
Доцент кафедры аэрокосмических компьютерных и программных систем, кандидат технических наук
Санкт-Петербургский государственный университет аэрокосмического приборостроения
Россия, 190000, Санкт-Петербург, ул. Большая Морская, д.67, лит. А
E-mail: shakhomirov@guap.ru

INFORMATION ABOUT THE AUTHORS

Shakhomirova Natalia Evgenievna
Assistant of the Department of Aerospace Computer and Program Systems
Saint-Petersburg State University of Aerospace Instrumentation
67, Bolshaya Morskaia str., Saint-Petersburg, 190000, Russia
E-mail: n.shah2024@yandex.ru

Shakhomirov Andrey Viktorovich
Associate Professor of the Department of Aerospace Computer and Program Systems, PhD.
Saint-Petersburg State University of Aerospace Instrumentation
67, Bolshaya Morskaia str., Saint-Petersburg, 190000, Russia
E-mail: shakhomirov@guap.ru