Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Операционные системы и системное программирование

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

по теме

ПОТОКИ ИСПОЛНЕНИЯ, ВЗАИМОДЕЙСТВИЕ И СИНХРОНИЗАЦИЯ

БГУИР КР 1-40 02 01 001 ЛР

Выполнил: студент группы 250541,
Бобрик В. Ю.

Проверил: ст.преподаватель каф. ЭВМ,
Поденок Л. П.

Минск 2025

СОДЕРЖАНИЕ

1 Условия лабораторной работы

Здесь две задачи. Обе «производители-потребители» для потоков.

Изучаемые системные вызовы: `pthread_create`(), `pthread_exit`(), `pthread_join`(), `pthread_yield`(), `pthread_cancel`(), `pthread_cond_init`(), `pthread_cond_destroy`(), `pthread_cond_*wait`(), `pthread_cond_signal`().

5.1) Аналогична лабораторной `No` 4, но только с потоками, семафорами и мьютексом в рамках одного процесса.

Дополнительно обрабатывается еще две клавиши – увеличение и уменьшение размера очереди. Следует предусмотреть обработку запроса на уменьшение очереди таким образом, чтобы при появлении пустого места уменьшался размер очереди, а не очередной производитель размещал там свое сообщение.

5.2 Аналогична лабораторной `No` 5.1, но с использованием условных переменных.

2 Описание алгоритмов и решений

2.1 Задача №1

Алгоритм работы производителя:

Шаг 1. Генерация сообщения.

Шаг 2. Ожидание возможности добавить сообщение.

Шаг 3. Добавление сообщения в очередь.

Шаг 4. Отправка сигнала потребителям о том, что появилось новое сообщение.

Алгоритм работы потребителя:

Шаг 1. Ожидание наличия сообщения.

Шаг 2. Извлечение сообщения.

Шаг 3. Проверка контрольной суммы.

Шаг 4. Отправка сигнала производителям о том, что появился свободный слот.

Для обеспечения корректного доступа к общей очереди и минимизации состояния гонки используется следующий набор механизмов:

`queue.mutex` защищает доступ к данным очереди.

`resize_mutex` обеспечивает атомарность операций изменения размера очереди, предотвращая одновременное изменение структуры данных.

`sem_free` управляет количеством свободных слотов и позволяет производителям дождаться появления места в очереди.

`sem_full` (предназначенный для потребителей) служит для ожидания появления новых сообщений.

2.2 Задача №2

Алгоритм работы производителя:

Шаг 1. Инициализация потока: при запуске поток получает свой идентификатор через аргумент.

Шаг 2. Создаётся объект `message`. Случайным образом выбирается символ для типа сообщения (от 'A' до 'Z').

Шаг 3. Генерируется случайное число `r` для определения длины сообщения. Если `r` равно 256, то это задаётся как 256 байт, иначе – значение `r`.

Шаг 4. Заполнение данных. В массив `data` записываются случайные символы для первых `actual_len` байт, оставшиеся заполняются нулями.

Шаг 5. Вычисление контрольной суммы: Вызывается функция `calculate_hash(&m)`, результат сохраняется в `m.hash`.

Шаг 6. Добавление сообщения в очередь. Блокируется мьютекс очереди.

Шаг 7. Если очередь заполнена, поток ждёт сигнала условной переменной `not_full`, пока не появится свободное место.

Шаг 8. После появления места сообщение копируется в позицию хвоста кольца и обновляются счётчики.

Шаг 9. Поток посылает сигнал `not_empty`, чтобы пробудить потоки-потребители, и разблокирует мьютекс.

Шаг 10. На экран выводится сообщение о добавлении.

Шаг 11. Завершение работы: Как только переменная `terminate_flag` становится истинной, поток выходит из цикла, сообщает о завершении работы и завершает выполнение.

Алгоритм работы потребителя:

Шаг 1. Инициализация потока: при запуске поток получает свой идентификатор через аргумент.

Шаг 2. Извлечение сообщения. Блокируется мьютекс очереди.

Шаг 3. Если очередь пуста, поток ждёт сигнала условной переменной `not_empty`.

Шаг 4. После появления элемента, сообщение извлекается из позиции головы кольцевого буфера и обновляются счётчики.

Шаг 5. Поток посылает сигнал условной переменной `not_full` для пробуждения потоков-производителей, и разблокирует мьютекс.

Шаг 6. Обработка сообщения. На экран выводится информация о полученном сообщении.

Шаг 7. Проверка корректности сообщения. Вычисляется контрольная сумма и сравнивается с сохранённой в `m.hash`. В зависимости от результата выводится сообщение об ошибке или подтверждении корректности.

Шаг 8. Завершение работы: Как только переменная `terminate_flag` становится истинной, поток выходит из цикла, сообщает о завершении работы и завершает выполнение.

Для обеспечения корректного доступа к общей очереди и минимизации состояния гонки используется следующий набор механизмов:

Все функции, которые работают с очередью (например, `push_message` и `pop_message`), оборачивают доступ к общим данным вызовами `pthread_mutex_lock()` и `pthread_mutex_unlock()`.

Условные переменные для управления ожиданием:

`not_empty` – потоки-потребители ждут, когда в очереди появится хотя бы один элемент.

`not_full` – потоки-производители ждут, когда в очереди освободится место для новой записи

Флаг `terminate_flag` используется для корректного завершения работы потоков. Потоки периодически проверяют его значение, и если устанавливается требование о завершении (при вводе команды `q`), они выходят из своих циклов ожидания и завершают выполнение.

При изменении состояния очереди или при завершении работы (например, при выполнении команды `q`) используются вызовы `pthread_cond_broadcast()`. Это разбудит все потоки, ожидающие на условных переменных, позволяя им пересмотреть условие ожидания и корректно завершиться или продолжить работу после изменения состояния очереди.

В функциях работы с очередью предусмотрены проверки (например, при уменьшении размера очереди), что тоже способствует избежанию состояний гонки, связанных с некорректными изменениями размера очереди или доступа к данным, которые могли быть изменены другим потоком.

3 Функциональная структура проекта

3.1 Задача №1

Модуль `common` определяет константы и структуры, используемые по всему проекту

Макросы: INITIAL_QUEUE_SIZE (начальный размер очереди), MAX_QUEUE_SIZE (максимально допустимое число сообщений) и ключи для системных ресурсов.

Структура message: описывает сообщение с полями: тип сообщения (один символ), контрольная сумма, длина данных (где 0 значит 256 байт) и сам массив данных.

```
typedef struct {
  char type;
  unsigned short hash;
  unsigned char size;
  char data[256];
} message;
```

Структура message_queue: структура очереди сообщений.

```
typedef struct {
  int capacity;
  message buffer[MAX_QUEUE_SIZE];
  int head;
  int tail;
  int added_count;
  int removed_count;
  int free_slots;
} message_queue;
```

Модуль producer_consumer содержит прототипы функций для потоков-производителя и потоков-потребителя, а также прототип функции calculate_hash, вычисляющей контрольную сумму сообщений.

producer_thread() – прототип потока производителя.

consumer_thread() – прототип потока потребителя.

calculate_hash() – функция вычисления контрольной суммы.

Модуль thread_queue определяет структуру thread_message_queue – динамическую очередь сообщений, предназначенную для многопоточного доступа.

```
typedef struct {
  Message *buffer;       // динамический массив сообщений
  int capacity;          // текущая ёмкость очереди
  int head;              // индекс извлечения
  int tail;              // индекс добавления
  int count;             // число сообщений в очереди
  int added_count;       // общий счётчик добавленных сообщений
  int removed_count;     // общий счётчик извлечённых сообщений
  pthread_mutex_t mutex;   // мьютекс для защиты очереди
  sem_t sem_free;        // семафор для свободных слотов
  sem_t sem_full;        // семафор для заполненных слотов
} ThreadMessageQueue;
```

init_thread_queue() – инициализация очереди.

destroy_thread_queue() – освобождение ресурсов очереди.

resize_thread_queue() – изменение ёмкости очереди с перераспределением памяти и корректировкой значений семафоров.

3.2 Задача №2

Модуль common определяет константы и структуры, используемые по всему проекту.

Структура message описывает одно сообщение.

```
typedef struct {
  char type;           // Тип сообщения (один символ)
  unsigned short hash;     // Контрольная сумма сообщения
  unsigned char size;      // Если 0, то означает 256 байт
```

```
  char data[DATA_SIZE];    // Данные сообщения
} message;
```

Модуль `producer_consumer` реализует функциональность потоков-производителя и потоков-потребителя, а также предоставляет функцию расчета контрольной суммы для сообщений.

`producer_thread()` – поток-производитель. Генерирует случайное сообщение с рандомными данными, вычисляет его контрольную сумму и вставляет сообщение в очередь, после чего выводит информацию о произведенной операции.

`consumer_thread()` – поток-потребитель. Извлекает сообщение из очереди, проверяет корректность его контрольной суммы и выводит статус обработки.

`calculate_hash()` – вычисляет контрольную сумму сообщения.

Модуль `queue` реализует динамическую (кольцевую) очередь для обмена сообщениями между потоками, обеспечивая потокобезопасный доступ посредством мьютексов и условных переменных.

Структура `message_queue` описывает очередь.

```
typedef struct {
  Message *buffer;      // Динамический массив сообщений
  int capacity;         // Текущая ёмкость очереди
  int count;            // Число сообщений в очереди
  int head;             // Индекс для извлечения
  int tail;             // Индекс для вставки
  int added_count;      // Счётчик добавленных сообщений (для отладки)
  int removed_count;    // Счётчик извлечённых сообщений (для отладки)
  pthread_mutex_t mutex;  // Мьютекс для защиты очереди
  pthread_cond_t not_empty; // Условная переменная: очередь не пуста
```

pthread_cond_t not_full; // Условная переменная: очередь имеет свободное место

} message_queue;

init_queue() – инициализирует очередь: выделяет память для буфера, устанавливает начальные значения, инициализирует мьютекс и условные переменные.

destroy_queue() – освобождает память, уничтожает мьютекс и условные переменные, выполняя очистку очереди.

push_message() – добавляет сообщение в очередь.

pop_message() – извлекает сообщение из очереди.

resize_queue() – изменяет размер очереди, выделяя новый буфер и копируя текущие элементы с учётом кольцевой структуры, а также проверяет возможность уменьшения.

print_status() – выводит текущее состояние очереди: ёмкость, количество элементов, а также отладочные счётчики.

Модуль приложения main является точкой входа в приложение, управляет созданием потоков, обработкой пользовательских команд и корректным завершением работы программы.

main() – основной цикл программы. Инициализирует очередь с использованием init_queue() и генератора случайных чисел, выводит команды для управления, создает новые потоки; при завершении работы устанавливает флаг terminate_flag, производит широковещательную рассылку сигналов на условные переменные, ожидает завершения всех созданных потоков, очищает ресурсы перед завершением работы программы.

4 Порядок сборки и тестирования

Для сборки используется Makefile, содержащий следующие ключевые элементы:

– Исходные файлы находятся в каталоге ./src.

– Заголовочные файлы расположены в каталоге ./include.

– В зависимости от типа сборки создаётся каталог ./out/debug для debug-версии или ./out/release для release-версии.

– Переменная BUILD задаётся как debug по умолчанию. При запуске make release устанавливается оптимизированная сборка.

– Флаги компиляции задаются через переменные COMMON_CFLAGS (определения, предупреждения, стандарт C11) и дополняются флагами отладки (-g -O0) или оптимизации (-O2).

Для сборки в debug-режиме необходимо выполнить в терминале:

```
make
```

Это соберёт программу в ./out/debug/.

Для сборки в release-режиме необходимо выполнить в терминале:

```
make release
```

Это соберёт программу в ./out/release/.

Для очистки сборки необходимо выполнить в терминале:

```
make clean
```

Тестирование происходит в интерактивном режиме во время выполнения программы. Программа предоставляет интерфейс, через который можно создавать потоки-производителей и потоков-потребителей, изменять размер очереди и получать текущий статус системы.

5 Методика и результаты тестирования

После сборки проекта необходимо запустить программу. Она выведет список доступных команд.

```
$ out/release/threads
Команды:
  + : добавить поток-производитель
  - : добавить поток-потребитель
```

```
> : увеличить размер очереди
< : уменьшить размер очереди
p : вывести состояние очереди
q : завершение работы
```

Программа обрабатывает следующие команды:

+ – создание нового потока-производителя. Ожидаемый результат: начало работы производителя.

```
$ +
Producer[1]: добавлено сообщение (тип 'W', размер 35, hash 2785). Всего добавлено: 1
Producer[1]: добавлено сообщение (тип 'P', размер 254, hash 19793). Всего добавлено: 2
Producer[1]: добавлено сообщение (тип 'O', размер 232, hash 18142). Всего добавлено: 3
```

- – создание нового потока-потребителя. Ожидаемый результат: начало работы потребителя.

```
$ -
Consumer[1]: извлечено сообщение (тип 'X', размер 135, hash 10301). Всего извлечено: 1
Consumer[1]: сообщение корректно.
Producer[1]: добавлено сообщение (тип 'D', размер 116, hash 8902). Всего добавлено: 2
Consumer[1]: извлечено сообщение (тип 'D', размер 116, hash 8902). Всего извлечено: 2
Consumer[1]: сообщение корректно.
Producer[1]: добавлено сообщение (тип 'B', размер 37, hash 2870). Всего добавлено: 3
Consumer[1]: извлечено сообщение (тип 'B', размер 37, hash 2870). Всего извлечено: 3
Consumer[1]: сообщение корректно.
```

p – вывод состояния очереди. Ожидаемый результат: Отображаются текущая ёмкость очереди, число элементов в очереди, количество свободных

слотов, а также статистика по количеству добавленных и извлечённых сообщений.

```
$ p
--- Состояние очереди ---
Ёмкость очереди: 10
Элементов в очереди: 1
Свободных слотов: 9
Добавлено сообщений: 10
Извлечено сообщений: 9
```

> – увеличение размера очереди на один слот. Ожидаемый результат: После успешного выполнения программа сообщает об изменении ёмкости очереди и обновлённом количестве свободных мест.

```
$ >
Resize: old_capacity=10, new_capacity=11, count=1, delta free=1
Размер очереди изменен: новая ёмкость = 11 (занято 1)
Очередь увеличена до 11 слотов
```

< – уменьшение размера очереди на один слот. Ожидаемый результат: Уменьшение ёмкости производится, только если очередь пуста и новая ёмкость не ниже минимально допустимой (2 слота). При соблюдении условий программа подтверждает успешное изменение.

```
$ <
Resize: old_capacity=11, new_capacity=10, count=1, delta free=-1
Размер очереди изменен: новая ёмкость = 10 (занято 0)
Очередь уменьшена до 10 слотов
```

`q` – завершение работы программы. Ожидаемый результат: Программа устанавливает флаг завершения, ожидает корректное завершение всех потоков (с использованием `pthread_join`), освобождает ресурсы и завершает работу.

```
Producer[1] завершает работу
Consumer[1] завершает работу
Программа завершена.
```