ОТЧЕТНАЯ РАБОТА НОМЕР 1.

ЯЗЫК РАЗРАБОТКИ **C/C**++.

ТЕМА: АНАЛИЗ ПОТОКОВ/СИГНАЛОВ СПЕЦИАЛИЗИРОВАННОЙ СИСТЕМЫ ОПЦИОНАЛЬНЫХ ДАННЫХ.

2024г.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ВЫПОЛНЕНИЮ РАБОТЫ.

1. Подключение необходимых библиотек.

Библиотека pthread (POSIX Threads) является стандартной библиотекой для работы с потоками в POSIX-совместимых операционных системах, таких как UNIX и Linux. Она предоставляет набор функций и типов данных для создания, управления и синхронизации потоков выполнения в многопоточных приложениях на C и C++.

```
#include <pthread.h>
#include <iostream>
#include <unistd.h>
#include <signal.h>
```

2. Теория: Определения и терминология.

Поток (Thread) - В многопоточном программировании, поток (или нить выполнения) представляет собой наименьшую единицу обработки, которая может быть запущена и выполнена независимо от других потоков в рамках одного процесса. Основные характеристики потока включают:

- **Параллельное выполнение**: Потоки могут выполняться параллельно на многопроцессорной архитектуре или конкурентно на однопроцессорной, совместно используя выделенные ресурсы.
- **Состояния потока**: Поток может находиться в различных состояниях, таких как выполнение (running), готовность (ready), блокировка (blocked) и т.д., в зависимости от его текущей активности и доступности ресурсов.
- **Синхронизация**: Потоки могут совместно использовать общие данные и ресурсы, что требует правильной синхронизации для избежания состояний гонки и обеспечения корректности данных.

Параллельное выполнение: Это понятие описывает ситуацию, когда несколько задач выполняются одновременно. В языке C++, параллельное выполнение достигается за счет запуска нескольких потоков, каждый из которых выполняет часть программы независимо от других.

Параллелизм: Параллелизм подразумевает фактическое одновременное выполнение нескольких задач на нескольких процессорах или ядрах процессора. В отличие от параллельного выполнения, которое может быть достигнуто и на однопроцессорной системе путем переключения контекста между потоками, параллелизм требует физической возможности одновременно обрабатывать несколько инструкций.

Для работы с потоками в C++ используются различные библиотеки и API. Одним из распространённых подходов является использование библиотеки pthreads (POSIX Threads), которая предоставляет средства для работы с потоками в POSIX-совместимых операционных системах, таких как Linux, macOS и другие UNIX-подобные системы.

основные функции и типы данных из библиотеки pthreads:

pthread_t - это тип данных, представляющий собой идентификатор потока в POSIX Threads. Каждый поток имеет свой уникальный pthread_t, который используется для идентификации и управления потоком.

pthread_attr_t: Тип, представляющий атрибуты потока, такие как размер стека, приоритет и другие параметры.

pthread_mutex_t, pthread_rwlock_t, pthread_cond_t: Типы данных для реализации мьютексов, семафоров и условных переменных для синхронизации доступа к ресурсам между потоками.

pthread_create - функция, используемая для создания нового потока в POSIX Threads.

pthread_exit - функция используется для завершения выполнения потока и возврата результата.

pthread_join - функция используется для ожидания завершения выполнения потока.

pthread_detach - функция используется для указания, что поток должен быть автоматически освобожден (detached) после завершения выполнения.

pthread_kill - функция используется для отправки сигнала определённому потоку.

Основные виды сигналов:

- **SIGINT** (2) - сигнал, который отправляется при нажатии комбинации клавиш Ctrl+C. По умолчанию приводит к завершению процесса.
- **SIGQUIT** (3) - сигнал, который отправляется при нажатии комбинации клавиш Ctrl+\ (Ctrl+Backslash). Обычно используется для запроса завершения процесса с возможностью получения core dump'a.
- **SIGKILL** (9) - сигнал, который немедленно завершает процесс. Он не может быть перехвачен или заблокирован процессом, и поэтому используется как последнее средство для принудительного завершения процесса.
- **SIGTERM** (15) - сигнал завершения, который по умолчанию отправляется командой kill. Этот сигнал может быть перехвачен процессом для выполнения специфических действий перед завершением.
- **SIGSTOP** (19) - сигнал, который останавливает выполнение процесса. Он также не может быть перехвачен или обработан процессом.
- **SIGCONT** (18) - сигнал, который возобновляет выполнение процесса, который был остановлен сигналом SIGSTOP или посылкой SIGTSTP.
- **SIGSEGV** (11) - сигнал, который отправляется при попытке доступа к недопустимой области памяти (нарушение сегментации).
- **SIGILL** (4) - сигнал, который отправляется при попытке выполнения недопустимой инструкции процессора.
- **SIGBUS** (10) - сигнал, который отправляется при ошибке доступа к физической памяти, такой как невыровненное обращение или ошибка шины.
- **SIGUSR1** (30) и **SIGUSR2** (31) - пользовательские сигналы, которые могут использоваться приложениями для определённых целей.

3. Методики расчета.

1) ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА ПОТОКА;

Для реализации параллельной обработки потоков и оптимизации системы, можно использовать условные переменные (condition variables) и мьютексы (mutexes). Это позволит избежать активного ожидания (busy-waiting) в потоках, что улучшит производительность и эффективность программы.

Добавим для этого в наш код константы:

```
pthread_mutex_t signalMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t signalCond = PTHREAD_COND_INITIALIZER;
```

измененный обработчик SIGUSR:

```
void handleSignal(int sig) {
    pthread_mutex_lock(&signalMutex);
    if (sig == SIGUSR1) {
        sharedSignal = 1; // Устанавливаем сигнал для второго потока
    } else if (sig == SIGUSR2) {
        sharedSignal = 2; // Устанавливаем сигнал для третьего потока
    }
    pthread_cond_broadcast(&signalCond);
    pthread_mutex_unlock(&signalMutex);
}
```

также поменяется код при ожидании сигнала:

```
void* threadFunction2(void* arg) {
    // Ждем сигнала от потока
    pthread_mutex_lock(&signalMutex);
    while (sharedSignal != 1) {
        pthread_cond_wait(&signalCond, &signalMutex);
    }
    pthread_mutex_unlock(&signalMutex);

    ...
    pthread_exit(nullptr);
}
```

Подготовка окружения разработки.

1. Установка необходимых инструментов для разработки на C/C++ (*например, компиляторы, IDE - Visual Studio, CLion, или другие*).
2. Подготовка библиотек для дальнейшей работы.

Требования.

1. Соответствие заявленному заданию и методическим указаниям.
2. Использование верных подходов к реализации проекта.
3. Соответствие нормам и стандартам кода C/C++.
4. Применение THREADS средств.
5. Применение Основ языка C/C++.
6. Правильное использование всех библиотек.
7. Описание функций (docstring).
8. Код должен содержать минимум 2 функции.
9. Отступы между функциями (2), между методами класса (1). Отступы между операторами, и тп.
10. Применение средств параллельного программирования (*при необходимости*)
11. Применение средств системного программирования.
12. Применение средств функционального программирования (*при необходимости*)
13. Применение WinAPI/QT (*при необходимости*)
14. Правильное оформление отчета согласно заявленным требованиям оформления.

1. ЗАДАНИЕ А

1.1 Изучение методических указаний, подключение и настройка необходимых библиотек для дальнейшей работы. Исследование определение/терминологии. Знакомство с потоками.

1.2 Выбрать ровно 2 действующих сигнала. Написать описание, что делает данный сигнал.

Пример таблицы:

Сигнал	**Описание**
SIGINT	….
…	и другие

**рекомендовано взять сигналы семейства SIGUSR*

1.3 Определить три потока с именами **thread1**, **thread2** и **thread3** соответственно.

1.4 Первый поток спит на N секунд (N приходит из arg), затем посылает сигнал второму потоку - после чего завершается. (*sleep - исп-ся для засыпания #include <unistd.h>*)

1.5 Второй поток выводит пришедший сигнал от первого потока, отправляет новый сигнал третьему потоку и завершается.

1.6 Третий поток принимает сигнал от второго потока, вычисляет произведение трех целых чисел - которые ему пришли через arg и завершается.

1.7 Сделать микровывод о работе с потоками. (*какой поток отработал быстрее всех?*)

2. ЗАДАНИЕ Б

2.1 Используя код из задания А. Реализовать параллельную обработку каждого потока, оптимизировав тем самым систему.
(*см Методика Расчета*)

2.2 Сравнить последовательную обработку с параллельной обработкой потоков.

2.3 Проделать аналогичную работу для сигналов другого семейства. Например, если вы сначала взяли SIGUSR, то теперь возьмите SIGINT/SIGILL. После сравните для каких сигналов лучше отрабатывает или какие сигналы можно использовать, а какие нельзя.

2.4 Сформулируйте микровывод.

ПОСЛЕ ВСЕЙ РАБОТЫ СФОРМУЛИРУЙТЕ ОБЩЕЕ-ТЕЗИСНОЕ ЗАКЛЮЧЕНИЕ.

ОБЯЗАТЕЛЬНО СОСТАВЬТЕ ОТЧЕТ О ПРОДЕЛАННОЙ РАБОТЕ СОГЛАСНО ТРЕБОВАНИЯМ ОТЧЕТНОЙ РАБОТЫ.

ПРИЛОЖЕНИЕ А.

ПРИМЕР КОДА - НУЖНЫЕ ПЕРЕМЕННЫЕ.

```
// переменные для передачи сигналов между потоками
volatile sig_atomic_t sharedSignal = 0;
pthread_t thread1, thread2, thread3;
```

ПРИМЕР КОДА - ОБРАБОТЧИК СИГНАЛА SIGUSR.

```
void handleSignal(int sig) {
    if (sig == SIGUSR1) {
        sharedSignal = 1; // установка сигнала для второго потока
    } else if (sig == SIGUSR2) {
        sharedSignal = 2; // установка сигнала для третьего потока
    }
}
```

ПРИМЕР КОДА - В MAIN (УСТАНОВКА ОБРАБОТЧИКОВ СИГНАЛА).

```
struct sigaction sa;
sa.sa_handler = handleSignal;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction(SIGUSR1, &sa, NULL);
sigaction(SIGUSR2, &sa, NULL);
```

ПРИМЕР КОДА - СОЗДАНИЕ ПОТОКА (СО ЗНАЧЕНИЕМ).

```
pthread_create(&thread, NULL, threadFunction, (void*)&value);
```

ПРИМЕР КОДА - ОЖИДАНИЕ ЗАВЕРШЕНИЕ ПОТОКА.

```
pthread_join(thread, NULL);
```

ПРИМЕР КОДА - ПОСЫЛАЕМ СИГНАЛ ПОТОКУ.

```
void* threadFunction(void* arg) {
    // Посылаем сигнал потоку
    pthread_kill(thread, SIGUSR1);
    pthread_exit(nullptr);
}
```

ПРИМЕР КОДА - ОЖИДАНИЕ СИГНАЛА ОТ ПОТОКА.

```
void* threadFunction2(void* arg) {
    // Ждем сигнала от потока
    while (sharedSignal != 1) sleep(1);
    ...
    pthread_exit(nullptr);
}
```