

Лабораторные работы по курсу
Операционные системы

Лабораторная работа 3
Потоки в операционной системе

Москва, 2023

Оглавление

Теоретическая часть.....	3
1. Понятие потока в ОС.....	3
2. Организация потоков в ОС Linux	4
3. Интерфейс OpenMP	5
Практическая часть	7
Контрольные вопросы	10
Список рекомендованной литературы	11

Теоретическая часть

1. Понятие потока в ОС

В предыдущей лабораторной работе было дано определения процесса. Напомним, что процесс - программа во время исполнения или объект, которому выделяются ресурсы вычислительной системы, такие как процессорное время, память и т. д. Каждый процесс, содержит в себе набор последовательно выполняемых инструкций. Такой набор называется **потоком** (*thread*). Не нужно путать его со стандартными потоками ввода-вывода (*streams*). Они отвечают за коммуникацию между процессами.

Каждый процесс как минимум состоит из одного потока. В настоящее время очень популярным решением является применение многопоточности в программировании – создание в рамках одного процесса нескольких потоков, работающих параллельно (квазипараллельно).

Представим следующую ситуацию. Нам необходимо написать программу, которая останавливает свою работу по нажатию кнопки. Если программа будет работать всего в один поток, то придется вместе с логикой программы производить опрос кнопки. Это не является эффективным решением, т. к. в момент нажатия кнопки, возможно выполнение кода программы, а не команды опроса. Таким образом нажатие будет пропущено. Гораздо более эффективным способом является разбиение программы на два потока – один отвечает за логику работы, а второй за работу с кнопкой.

В современном мире растет число процессоров, содержащих в себе несколько ядер. Применяя параллельное программирование, мы можем ускорить работу нашей программы в несколько раз! Однако, обратим внимание на то, что созданные потоки не обязательно будут выполняться параллельно на разных ядрах. На процессоре с одним ядром также возможна многопоточность – потоки будут исполняться квазипараллельно. Другими словами, часть процессорного времени будет выполняться один поток, часть – другой. Эти части настолько малы, что для пользователей создается иллюзия параллельности. Существует высказывание – «*Concurrency is not parallelism*». Если перевод слова *parallelism* не требует пояснений, то *Concurrency* обычно переводят как многопоточность. Т.е. «многопоточность \neq параллельность».

Все потоки создаются в рамках одного процесса. Таким образом, потоки делят одно адресное пространство. Если внутри процесса будет создана глобальная переменная, то она будет видна как из одного потока, так и из другого. Попытка одновременного изменения этой переменной несколькими потоками приведет к неопределенному состоянию. Неизвестно что произошло ранее – запись первым потоком или запись вторым. (Рисунок 1)

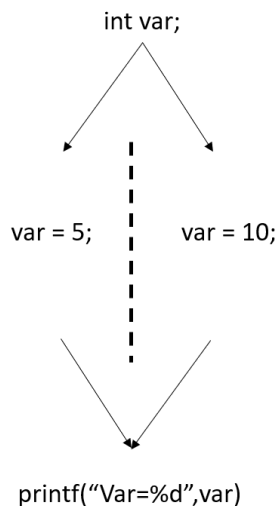


Рисунок 1 Неопределенное состояние

Ситуация, когда две инструкции из разных потоков, одна из которых – инструкция записи, пытаются получить доступ к одной ячейке памяти называется *гонкой данных* или *data race*. Пример, приведенный выше – пример гонки данных. Для безопасной работы с потоками существуют специальные механизмы синхронизации, речь о которых пойдет в следующей лабораторной работе.

2. Организация потоков в ОС Linux

Для создания переносимых многопоточных программ был разработан специальный стандарт *POSIX.1c*. Пакет, предназначенный для работы с потоками, называется *pthread*. Таким образом, потоки, созданные с помощью данного пакета, часто называют POSIX потоками. Он поддерживается большинством UNIX систем, однако по умолчанию не поддерживается Windows.

Для создания и управления потоками в C используются следующие основные функции из заголовочного файла «*pthread.h*»:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg) -  
функция создания потока;  
void pthread_exit(void *retval) - функция для завершения потока;  
int pthread_join (pthread_t THREAD_ID, void ** DATA) - функция для ожидания потока;
```

3. Механизмы синхронизации потоков

Для корректной работы многопоточной программы необходимо синхронизировать работу потоков. Для этого существуют несколько стандартных механизмов. Наиболее распространенным и популярным является мьютекс (*mutex*).

Mutex – (от англ. *mutual exclusion* — «взаимное исключение») - примитив синхронизации, обеспечивающий взаимное исключение исполнения критических участков кода. Мьютекс можно представить в виде переменной, которую можно перевести в одно из двух состояний – заблокировано и разблокировано. При входе в критическую секцию поток вызывает функцию перевода мьютекса в заблокированное состояние, при этом поток блокируется до освобождения мьютекса, если другой поток

уже владеет им. При выходе из критической секции поток вызывает функцию перевода мьютекса в незаблокированное состояние.

Для работы с мьютексами используются следующие функции:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr) - инициализация мьютекса
int pthread_mutex_lock(pthread_mutex_t *mutex) - захват мьютекса
int pthread_mutex_unlock(pthread_mutex_t *mutex) - освобождение мьютекса
```

Помимо обычных мьютексов используются рекурсивные мьютексы. Данный тип отличается от обычных мьютексов тем, что его возможно захватить несколько раз одним и тем же потоком. Инициализация в POSIX происходит следующим образом:

```
pthread_mutex_t Mutex;
pthread_mutexattr_t Attr;

pthread_mutexattr_init(&Attr);
pthread_mutexattr_settype(&Attr, PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&Mutex, &Attr);
```

4. Интерфейс OpenMP

OpenMP — это библиотека для параллельного программирования вычислительных систем с общей памятью. С ее помощью возможно без особых затрат распараллелить код программы на несколько потоков.

Для использования библиотеки OpenMP вам необходимо подключить заголовочный файл "omp.h", а также добавить опцию сборки -fopenmp (для компилятора gcc).

После запуска программы создается единственный процесс с единственным потоком, который начинается выполняться, как и обычная последовательная программа. Встретив параллельную область (задаваемую директивой #pragma omp parallel) процесс порождает ряд потоков (их число можно задать явно, однако по умолчанию будет создано столько потоков, сколько в вашей системе вычислительных ядер). Границы параллельной области выделяются фигурными скобками, в конце области потоки уничтожаются. В следующем примере программа выведет на экран сообщение внутри параллельной области несколько раз.

```
#include <stdio.h>
#include <omp.h>

int main()
{
    printf("Hello!\n");
    #pragma omp parallel
    {
        printf("I'm in the parallel section!\n");
    }
    printf("Goodbye!\n");
    return 0;
}
```

Результат работы программы:

```
os@os-VirtualBox:~/Desktop/lab3$ ./omp
Hello!
I'm in the parallel section!
I'm in the parallel section!
I'm in the parallel section!
I'm in the parallel section!
I'm in the parallel section!
I'm in the parallel section!
Goodbye!
```

Запуск данной программы производился на 6 ядерном процессоре. Таким образом, мы видим 6 записей в параллельной секции.

Практическая часть

В ходе выполнения практического задания вам необходимо провести эксперименты по расчету численного значения определенного интеграла при различных условиях.

В качестве интеграла предлагается взять следующий: $I = \int_0^1 \frac{4}{1+x^2} dx$

За число разбиений интеграла для вычисления примите значения 10^3 , 10^6 , 10^9 . Результаты экспериментов необходимо заносить в отчет. Полученное время выполнения программ рекомендуется свести в таблицу 1, приведенную ниже.

В отчет необходимо добавить графики из задания 3.

Также необходимо указать характеристики устройств, на которых производятся вычисления – название процессора, число ядер, тактовые частоты. Для этого возможно воспользоваться программой CPU-Z.

Для корректной работы на виртуальной машине необходимо включить возможность доступа к нескольким ядрам процессора. Для этого в настройках виртуальной машины (Настройки → Система → Процессор) выберите несколько ядер для работы.

4.1. Непосредственное вычисление интеграла.

Изучите приведенный ниже код:

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

//Input arguments:
//1: Начало интервала
//2: Конец интервала
//3: Число разбиений
//
//example:
// ./integral 0 10 1000000

double integrate(double a, double b, int n) {

    //тут будет ваш код

}

int main(int argc, char* argv[])
{
    double I;
    if (argc != 4) {
        fprintf(stderr, "Not enough arguments\n");
        exit(1);
    }
    long int N = atoi(argv[3]);
    I = integrate(strtod(argv[1], 0), strtod(argv[2], 0), N);
    printf("I = %f\n", I);

    return 0;
}
```

Создайте файл *integral.c* и скопируйте в него данную программу. Дополните функцию *rectangle_integral*, чтобы она возвращала значение посчитанного интеграла. Скомпилируйте проект и запустите его на виртуальной машине.

Для компиляции воспользуйтесь следующим ключом:

gcc integral.c -o integral -lm

Сверьте решение, полученное вашей программой с точным решением. (Например, полученным с помощью онлайн калькулятора)

С помощью утилиты `time` замерьте время выполнения вашей программы

time ./integral 0 10 1000000

Требуемое значение будет выведено в строке *real*.

4.2. Разбиение вычисления выполнения интеграла на несколько процессов

Изучите приведенный ниже код:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <math.h>

double integrate(double a, double b, double n) {

    //тут будет ваш код

}

int main(int argc, char* argv[])
{
    double I;
    if (argc != 4) {
        fprintf(stderr, "Not enough arguments\n");
        exit(1);
    }
    long int N = atoi(argv[3]);

    int n;
    int fds[2];
    pid_t pid;
    pipe(fds);
    pid = fork();
    if (pid != (pid_t)0)
    {
        double r1 = 0, r2 = 0;
        int s;
        close(fds[1]);
        r1 = integrate(strtod(argv[1], 0), strtod(argv[2], 0) / 2, N / 2);
        read(fds[0], &r2, sizeof(double));
        close(fds[0]);
        printf("I = %g\n", r1 + r2);
        wait(&s);
        return 0;
    }
    else
    {
        double s;
        close(fds[0]);
        s = integrate(strtod(argv[2], 0) / 2, strtod(argv[2], 0), N / 2);
        write(fds[1], &s, sizeof(double));
        close(fds[1]);
        return 0;
    }
}
```

Аналогично заданию 1 дополните код программы. Запустите ее и убедитесь, что значение рассчитано верно.

Замерьте время выполнения программы. Сравните с п.1 Сделайте выводы.

4.3. Вычисление интеграла с помощью POSIX threads

Изучите приведенный ниже код:

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

double integrate(double a, double b, double n) {

    //тут будет ваш код

}

struct IntegrateTask { // Шаблон для структуры "Задача потока"
    double from, to, step, res; // интегрировать "от" (from), "до" (to), с "шагом" (step),
    // результат сохранить в res
};

void* integrateThread(void* data) { // ф-я приведения типов задания и т.д.
    struct IntegrateTask* task = (struct IntegrateTask*)data; // объявления структуры task и
    // присвоения аргументов
    task->res = integrate(task->from, task->to, task->step); // вызов ф-и интегрирования с
    // передачей параметров (задача)

    pthread_exit(NULL); // завершение потока
}

int main(int argc, char* argv[]) {

    double I;
    if (argc != 5) {
        fprintf(stderr, "Not enough arguments\n");
        exit(1);
    }
    long int N = atoi(argv[3]);
    long int NUM_THREADS = atoi(argv[4]);
    pthread_t threads[NUM_THREADS]; // Объявляем массив структур потоков (системные)
    struct IntegrateTask tasks[NUM_THREADS]; // Объявляем массив структур заданий потокам

    struct IntegrateTask mainTask = { strtod(argv[1],0),strtod(argv[2],0),N / NUM_THREADS }; //
    // Общее задание, интегрировать от 0 до 10 с шагом 0.0000001
    double distance = (mainTask.to - mainTask.from) / NUM_THREADS; // Делим общее задание на
    // части

    int i;

    for (i = 0; i < NUM_THREADS; ++i) // создаем задания и потоки
    {
        tasks[i].from = mainTask.from + i * distance; // задаем "от"
        tasks[i].to = mainTask.from + (i + 1) * distance; // задаем "до"
        tasks[i].step = mainTask.step; // задаем "шаг"

        pthread_create(&threads[i], NULL, integrateThread, (void*)& tasks[i]); // создание
        // потоков и передача параметров (задания)
    }

    double res = 0;
    for (i = 0; i < NUM_THREADS; ++i)
    { // Барьер
        pthread_join(threads[i], NULL); // ждем завершения потока
        res += tasks[i].res; // суммируем результаты
    }
    printf("I = %lf \n", res);

    return 0;
}
```

Аналогично заданию 1 дополните код программы.

Скомпилируйте ее с помощью следующей команды:

gcc integral.c -o integral -lm -lpthread

Запустите ее и убедитесь, что значение рассчитано верно.

*Рассчитайте время выполнения программы при различном числе потоков – от 1 до 8. Постройте **график** зависимости время выполнения от числа потоков.*

Сравните время выполнения программы, выполняемой двумя потоками с программой из задания 2. Объясните полученный результат.

4.4. Использование директив OpenMP

Исправьте код программы из задания 1, добавив в функцию `integrate` перед циклом `for` следующую строчку:

```
#pragma omp parallel for reduction(+:sum)
```

Где `sum` – накапливаемая сумма при расчете интеграла.

Скомпилируйте программу с помощью следующей команды:

gcc integral.c -o integral -lm -fopenmp

Запустите программу и вычислите время выполнения. Сравните его со временем выполнения программы предыдущих заданий.

Таблица 1. Сравнение времени выполнения программ

Задание	Время выполнения (сек)		
	n=10 ³	n=10 ⁶	n=10 ⁹
Последовательное выполнение (задания 1)			
Выполнение двумя процессами (задание 2)			
Выполнение с помощью POSIX threads (задание 3)			
Число потоков			
1			
2			
3			
4			
5			
6			
7			
8			
Выполнение с помощью OMP (задание 4)			

Контрольные вопросы

1. Какие основные отличия между процессами и потоками?
2. В чем преимущества использования потоков?
3. Для чего используется интерфейс OpenMP?

Список рекомендованной литературы

1. Э. Таненбаум и Х. Бос, Современные операционные системы, Санкт-Петербург: Питер, 2021, с. 1119.
2. В. Е. Карпов и К. А. Коньков, Основы операционных систем, Москва: Физматкнига, 2019, с. 326.
3. Учебник по OpenMP [Электронный ресурс] // Блог программиста: [сайт]. [2018]. URL: <https://pro-prof.com/archives/4335> (дата обращения: 12.09.2022).