ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

КАФЕДРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

ЛАБОРАТОРНАЯ РАБОТА № 3

«ПАРАЛЛЕЛЬНОЕ ВЫПОЛНЕНИЕ ОПЕРАЦИЙ НА УРОВНЕ ОПЕРАЦИОННОЙ СИСТЕМЫ. ПОТОКИ»

Автор: С.Н. Мамойленко

Новосибирск - 2016

Оглавление

Цель работы

Лабораторная работа направлена на развитие следующих общекультурных, общепрофессиональных и профессиональных компетенций:

- способностью к профессиональной эксплуатации современного оборудования и приборов (ОК-8);
- владением методами и средствами получения, хранения, переработки и трансляции информации посредством современных компьютерных технологий, в том числе в глобальных компьютерных сетях (ОПК-5);
- способностью проектировать системы с параллельной обработкой данных и высокопроизводительные системы, и их компоненты (ПК-9);
- способностью к программной реализации систем с параллельной обработкой данных и высокопроизводительных систем (ПК-14).

В результате изучения дисциплины должно быть сформировано понимание принципов организации параллельного выполнения программ с использованием потоков выполнения и автоматизации распараллеливания программ с использованием стандарта OpenMP, а также получены умения и навыки по проектированию и разработке параллельного программного обеспечения с использованием потоков выполнения и технологии OpenMP.

Теоретическое введение

1. Базовые сведения о легковесных процессах (нитях)

Многозадачная операционная система может выполнять несколько программ одновременно, при этом каждая программа может иметь несколько потоков команд, которые называются нитями. Каждая нить представляет собой независимо выполняющуюся функцию со своим счетчиком команд, регистровым контекстом и стеком. Все нити выполняются в едином адресном пространстве. Нити часто называют легковесными процессами.

2. Создание потока

Поток выполнения (далее поток, нить) – это способ выполнения некоторой функции программы. «Создать поток» - значит сообщить операционной системе, какую из функций программы требуется выполнять и как при этом производить планирование использования ресурсов при выполнении нескольких потоков.

В операционной системе запуск программы на выполнение приводит к созданию одного потока, который предназначен для выполнения какой-то из функций программы (в языке Си такой функцией является `main`). Создание дополнительных потоков по стандарту POSIX осуществляется с использованием функции `pthread_create()`, заголовок которой представлен на рисунке 1.

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void* (*start)(void *), void *arg)
```

Рисунок 1 – Описание функции `pthread_create()`

Первый аргумент этой функции `thread` - это указатель на переменную типа `pthread_t`, в которую будет записан идентификатор созданного потока.

Второй аргумент `attr` задает набор некоторых свойств создаваемой нити, описывающий правила планирования исполнения нити, её окружение (стек и т.п.) и поведение нити при завершении (удалять ли информацию сразу или ожидать вызова `pthread_join`). Если требуется запустить нить с параметрами «по умолчанию», то в качестве этого параметра следует передать `NULL`.

Аргумент `start` - это указатель на функцию, которую надо выполнить в отдельном потоке. Именно эту функцию и начинает выполнять вновь созданная нить, при этом в качестве параметра этой функции передается четвертый аргумент вызова `pthread_create`.

Функция `pthread_create` возвращает нулевое значение в случае успеха и ненулевой код ошибки в случае неудачи. Пример использования функции `pthread_create` приведён на рисунке 2.

```
1  #include <stdio.h>
2  #include <pthread.h>
3  void * my_thread (void * arg){
4    char ch;
5    ch = *((char *)arg);
6    while (1) printf ("%c", ch);
7    return (NULL);
8  }
9  int main(void)
10 {
11   pthread_t thread_id1, thread_id2;
12   char ch1 = 'A', ch2 = 'B', ch3 = 'C';
13   pthread_create (&pthread_id1, NULL, my_thread, (void *)&ch1);
14   pthread_create (&pthread_id2, NULL, my_thread, (void *)&ch2);
15   my_thread ((void *)&ch3);
16   return (0);
17 }
```

Рисунок 2 – Пример программы, использующей три нити

В строках 1-2 подключаются необходимые заголовочные файлы. В данном случае нам требуются функции ввода-вывода (`<stdio.h>`) и функции по работе с нитями (`<pthread.h>`)[1]. Далее, в строках 3-8 описывается функция, которая будет дальше использоваться как тело отдельной нити. Создание нитей и запуск их на выполнение происходит в строках 13-14. Основная нить также выполняет функцию `my_thread` (строка 15). В итоге на экран терминала будет выведена последовательность из чередующихся символов A, B и C (каждая из нитей непрерывно выводит «свой» символ).

Следует обратить внимание, что каждая из нитей использует собственную локальную переменную `ch`. Забегая вперёд скажем, что приведённый пример не совсем корректен, так как функция `printf` из стандартной библиотеки `glibc` не является «безопасной для нитей».

3. Завершение нити

Нить завершается, когда происходит возврат из соответствующей ей функции или с использованием вызова `pthread_exit()`. Заголовок функции `pthread_exit()` приведен на рисунке 3.

```
#include <pthread.h>
int pthread_exit(void *value_ptr);
int pthread_cancel(pthread_t thread);
int pthread_kill (pthread_t thread, ..);
```

Рисунок 3 – Заголовки функций завершения нитей

Этот вызов завершает выполняемую нить, возвращая в качестве результата ее выполнения `value_ptr`.

[1] Следует помнить, что для компиляции приведённого примера следует указать ключ `-lpthread`, который определяет, что используется дополнительная библиотека `pthread` по работе с нитями.

Следует помнить, что завершение процесса всегда сопровождается завершением всех его нитей. Т.е. если нить выполнить вызов `exit` (а не `pthread_exit`), то будут завершены все нити и сам процесс.

Нить может быть завершена принудительно. Для отправки требования о завершении нити используется вызов `pthread_cancel()` или `pthread_kill()` (см. рисунок 3).

4. Ожидание завершения потока. Синхронизация выполнения потоков.

Чтобы узнать код завершения нити (какое значение будет возвращено её оператором `return`) используется функция `pthread_join`, заголовок которой представлен на рисунке 4.

```
#include <pthread.h>
int pthread_join(pthread_t thread, void** value_ptr)
```

Рисунок 4 – Заголовок функции `phtread_join()`

Эта функция дожидается завершения потока с идентификатором `thread`, и записывает его возвращаемое значение в переменную, на которую указывает `value_ptr`.

Для организации взаимоисключения стандарт POSIX предлагает использовать так называемый `mutex` (от mutual exclusion). В программах `mutex` представляется переменными типа `pthread_mutex_t`. Прежде, чем начать использовать объект типа `pthread_mutex_t`, необходимо провести его инициализацию. Это можно сделать при помощи функции `pthread_mutex_init ()` (см. рисунок 5).

```
#include <pthread.h>
int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Рисунок 5 – Заголовки функций работы с мьютексами

В случае если мы хотим создать `mutex` с атрибутами по умолчанию, то мы можем воспользоваться макросом `PTHREAD_MUTEX_INITIALIZER`.

После того как мы проинициализировали `mutex`, мы можем захватить его при помощи одной из функций: `pthread_mutex_lock ()` и `pthread_mutex_trylock()` (см. рисунок 5).

Первая из двух функций просто захватывает `mutex`, при этом, если на данный момент `mutex` уже захвачен другой нитью, эта функция дождется его освобождения. Вторая же функция попытается захватить `mutex` если он свободен, а если он окажется занят, то немедленно возвратит специальный код ошибки `EBUSY`. То есть `pthread_mutex_trylock` фактически является неблокирующим вариантом вызова `pthread_mutex_lock`.

При этом надо заметить, что нить, которая уже владеет `mutex`, не должна повторно пытаться захватить `mutex`, так как при этом либо будет возвращена ошибка, либо может произойти то, что в англоязычной литературе называется self-deadlock (самотупиковая ситуация), то есть нить будет ждать освобождения `mutex` до тех пор, пока сама не освободит его, то есть фактически до бесконечности.

Для освобождения захваченного mutex'а предназначена функция `pthread_mutex_unlock()`.

Стоит еще раз подчеркнуть, что нить может освобождать `mutex` только предварительно захваченный этой же нитью. Результат работы функции `pthread_mutex_unlock`() при попытке освободить захваченный другой нитью или вообще свободный `mutex` не определен. Другими словами вызов этой функции корректен если данная нить перед этим успешно выполнила либо

pthread_mutex_lock() или pthread_mutex_trylock() и не успела выполнить комплиментарный им pthread_mutex_unlock().

Процесс обязан рано или поздно (но в любом случае до повторного использования памяти, в которой располагается объект типа pthread_mutex_t) уничтожить его освободив связанные с ним ресурсы. Сделать это можно вызвав функцию pthread_mutex_destroy().

При этом на момент вызова этой операции уничтожаемый mutex должен быть свободен, то есть, не захвачен ни одной из нитей, в том числе вызывающей данную операцию. В случае если мы инициализировали mutex при помощи макроса PTHREAD_MUTEX_INITIALIZER, то необходимость в его явном уничтожении отсутствует.

5. Автоматизация разработки параллельного программного обеспечения. Технология OpenMP

Использование потоков выполнения используется в проектировании параллельного программного обеспечения, в котором необходимо обеспечить доступ к общим данным. Для автоматизации разработки такого программного обеспечения с использованием языков Си, Си++ и Фортран разработан стандарт (технология) OpenMP. Первая версия стандарта выпущена в 1997 году.

OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» поток создает набор подчиненных потоков и распределяет данные между ними. Задачи, выполняемые потоками параллельно, также, как и данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка.

Ключевыми элементами OpenMP являются:

- конструкции для создания потоков (директива parallel);
- конструкции распределения работы между потоками (директивы do/for и section);
- конструкции для управления работой с данными (выражения shared и private для определения класса памяти переменных);
- конструкции для синхронизации потоков (директивы critical, atomic и barrier);
- процедуры библиотеки поддержки времени выполнения (например, omp_get_thread_num);
- переменные окружения (например, OMP_NUM_THREADS).

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  #define N 100
5
6  int main(int argc, char *argv[]){
7    double a[N], b[N], c[N];
8    int i;
9    omp_set_num_threads(10); // установить число потоков в 10
10   for (i = 0; i < N; i++){ a[i] = i * 1.0; b[i] = i * 2.0; }
11   // вычисляем сумму массивов
12 #pragma omp parallel for shared(a, b, c) private(i)
13     for (i = 0; i < N; i++)
14       c[i] = a[i] + b[i];
15
16   printf ("%f\n", c[10]);
17   return 0;
18 }
```

Рисунок 6 – Пример программы, разработанной с использованием технологии OpenMP[2]

[2] Пример заимствован из свободной энциклопедии «Википедиа» (https://ru.wikipedia.org/wiki/OpenMP).

В языке Си все операции, которые должны выполнятся в соответствии с технологией OpenMP, описываются в виде директив препроцессора, начинающихся с `#pragma opm`. Для того, чтобы компилятор начал распознавать такие директивы, необходимо указать специальную опцию в командной строке при его запуске. Например, для компилятора GCC такой опцией является `–fopenmp`.

5.1 Создание параллельных секций. Директива parallel

Для того, чтобы компилятор создал группу из нескольких потоков используется директива `parallal` (см. рисунок 7).

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main(int argc, char *argv[]){
5  #pragma omp parallel num_threads (5)
6  {
7    printf (”Я – нить\n”);
8  }
9  return 0;
10 }
```

Рисунок 7 – Пример программы, использующей секцию `parallel`.

В приведённом на рисунке 7 примере директива `parallel` требует от компилятора создать 4 дополнительные нити (требуется всего 5 нитей, но одна уже есть) и в каждой из них выполнить одни и те же действия – выполнить функцию `printf`. Количество нитей может задаваться явно с использованием директивы `num_threads` (в этом случае в каждой секции нужно указать такую директиву) или использовать значение, устанавливаемое функцией `omp_set_num_threads()` или переменной среды окружения `OMP_NUM_THREADS`.

5.2 Распределение работы между нитями. Директива for.

Директива parallel требует от компилятора создать группу нитей и «заставить» каждую из них выполнять одни и те же действия. Для распределения обязанностей между нитями используются директив `for` (в языке Фортран – `do`).

Директива `for` предполагает распараллеливание цикла `for` путем распределения обрабатываемого циклом диапазона между нитями (например, если цикл предполагает выполнение действий с индексами от 0 до 9 и будет запущено две нити, то первая будет работать с индексами от 0 до 5, а вторая от 6 до 9).

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main(int argc, char *argv[]){
5    int i, t;
6  #pragma omp parallel num_threads (5)
7  {
8    t = omp_get_thread_num();
9  #pragma omp for
10   for (i = 0; i < 100; i++) { printf (”Нить %d. i = %d\n”, t, i); }
12 }
14 return 0;
15 }
```

Рисунок 8 – Пример программы, использующей директиву `for`.

Очевидно, что распараллелить цикл `for` возможно только в случае явного использования некоторой переменной в качестве счетчика итераций цикла. Если внутри параллельной секции выполняется только цикл `for`, то директивы `parallel` и `for` могут объединяться (как на рисунке 6).

5.3 Контроль доступа к данным. Директивы critical, private

Запуская нити на выполнение, компилятор оставляет все используемые в них данные в общем адресном пространстве и нити могут влиять таким образом друг на друга.

Если необходимо, чтобы какой-то участок кода выполнялся строго нитями строго последовательно (так называемая критическая секция), то используется директива `critical`.

Для того, чтобы сделать какие-либо переменные локальными для каждого потока в директиве `for` используется модификатор `private`.

В программе, представленной на рисунке 6, в строках 12 – 16 выполняется распараллеливание цикла `for` с указанием, что массивы являются разделяемыми (доступными во всех нитях), а переменная `i` становится для каждой нити локальной (если её изменить в одной нити, то в других нитях это значение не изменится).

Задание на лабораторную работу

Базовые задания.

Разработать программу, реализующую модель работы склада, отвечающего за хранение и продажу некоторого товара (одного). Склад содержит N помещений, каждый из которых может хранить определённое количество единиц товара. Поступающий товар помещается в одно из помещений специальным погрузчиком. За товаром прибило K покупателей, каждому из которых требуется по L_k единиц товара. Площадка перед складом мала и на ней может в один момент времени находиться либо погрузчик, либо один из покупателей. Если покупателям требуется больше товара, чем имеется на складе, то они ждут новых поступлений, периодически проверяя склад. Время работы склада ограниченно.

- Основная нить (функция main) выполняет следующие действия:
  - Формирует начальное заполнение склада (для каждого помещения случайным образом выбирается число из диапазона от 1 до 40);
  - Обрабатывает опции командной строки, в которой должно быть указано сколько клиентов будет обслуживаться складом и в течении какого времени должен склад работать;
  - Порождает заданное количество нитей, каждая из которых реализует алгоритм работы покупателя. Каждому покупателю случайным образом назначается количество требуемых единиц продукции (число из диапазона от 1 до 1000).
  - Настраивает таймер (alarm) таким образом, чтобы он сработал по окончании времени работы склада;
  - Запускает алгоритм работы погрузчика;
  - После срабатывания таймера принудительно завершает все выполняющиеся нити (если таковые имеются).
  - Завершает работу программы.
- Алгоритм работы погрузчика.
  - Пытается попасть на площадку перед складом;
  - Как только попадет на площадку, ищет хотя бы один склад, в котором нет продукции, и заполняет его максимально возможным образом;
  - покидает площадку;
  - «засыпает» на 5 секунд;
  - Цикл повторяется до срабатывания таймера;
- Алгоритм работы покупателя.
  - Пытается попасть на площадку перед складом;

- Как только попадет на площадку, ищет хотя бы один склад, в котором есть продукция, и забирает либо столько, сколько надо, либо всю продукцию;
- покидает площадку;
- «засыпает» на 5 секунд;
- Цикл повторяется до тех пор, пока покупателю нужна продукция;

Программа должна на экран выводить информацию о помещениях склада.

Основные задания.

Доработайте программу умножения матриц из лабораторной работы № 2 с наилучшим способом обхода оперативной памяти так, чтобы использовалось автоматическое распараллеливание циклов for. Продемонстрируйте, что результат умножения матриц получился правильным. Оцените получившееся ускорение выполнения программы.

Задания повышенной сложности.

Спроектируйте и разработайте параллельное приложение, реализующее игру «крестики-нолики».

Одна нить отвечает за интерактивное взаимодействие с пользователем. Ожидает ввод с клавиатуры определённых клавиш (остальные клавиши игнорируются). Если пользователь нажимает клавишу S, то нить сообщает второй нити, что можно начать или прекратить «играть» (см. ниже). Если нажата клавиша T, то пользователю предлагается ввести целое число в диапазоне от 1 до 10. После ввода первая нить меняет значение общей переменной `timeThink`. Если пользователь нажимает клавишу A, то ему предлагается ввести целое число в диапазоне от 5 до 30, которое записывается в общую переменную `timeRestart`.

Вторая нить реализует имитацию игры «Крестики-нолики» между двумя игроками. Каждый игрок «думает» в течение времени `timeThink` и делает свой ход случайным образом в любую свободную ячейку. Игра продолжается до тех пор, пока какой-то из игроков не выиграет или не будет заполнено все поле. После окончания игры происходит перезапуск после ожидания `timeRestart` секунд.

Контрольные вопросы

1. Что такое поток выполнения? Как потоки выполнения используются в многозадачных операционных системах?
2. Какие функции в стандарте POSIX используются для работы с потоками выполнения?
3. Имеет ли возможность один поток выполнения изменить данные другого потока выполнения?
4. Что такое критическая секция? Как она реализуется с использованием мьютексов?
5. Зачем используется технология OpenMP?
6. Что делают директивы `parallel`, `for`?
7. Зачем используются модификаторы `critical` и `private`?