**Лабораторная работа №5**

**Задание 1**

**Необходимые знания**

1.Компилирование программ с помощью gcc.

-Gcc– это свободно доступный оптимизирующий компилятор для языков C, C++.

-Программа gcc, запускаемая из командной строки, представяляет собой надстройку над группой компиляторов.

-В зависимости от расширений имен файлов, передаваемых в качестве параметров, и дополнительных опций, gcc  запускает необходимые препроцессоры, компиляторы, линкеры.

-Clang может использоваться как прямая замена gcc и предлагает несколько крутых инструментов статического анализа.

2.Состояние гонки.

-Ошибка программирования многозадачной системы, при которой работа системы зависит от того, в каком порядке выполняются части кода.

-Состояние гонки является классическим гейзенбагом.

-Состояние гонки возникает тогда, когда несколько потоков многопоточного приложения пытаются одновременно получить доступ к данным, причем хотя бы один поток выполняет запись. Состояния гонки могут давать непредсказуемые результаты, и зачастую их сложно выявить. Иногда последствия состояния гонки проявляются только через большой промежуток времени и в совсем другой части приложения. Кроме того, ошибки такого рода невероятно сложно воспроизвести повторно. Для предотвращения состояния гонки используются приемы синхронизации, позволяющие правильно упорядочить операции, выполняемые разными потоками.

3.Критическая секция.

-Критическая секция (Critical Section) это участок кода, в котором поток (thread) получает доступ к ресурсу (например переменная), который доступен из других потоков.

-Объект критическая секция обеспечивает синхронизацию. Этим объектом может владеть только один поток, что и обеспечивает синхронизацию. Для работы с критическими секциями есть ряд функций API и тип данных CRITICAL\_SECTION. Для использования критической секции нужно создать переменную данного типа, и проинициализировать ее перед использованием с помощью функции InitializeCriticalSection(). Для того, чтобы войти в секцию нужно вызвать функцию EnterCriticalSection(), а после завершения работы LeaveCriticalSection().

-Саму критическую секцию можно удалить функцией DeleteCriticalSection(). Для того, чтобы обойти блокировку потока при обращении к занятой секции есть функция TryEnterCriticalSection(), которая позволяет проверить критическую секцию на занятость.

4.POSIX threads: как создавать, как дожидаться завершения.

-POSIX определяет набор интерфейсов (функций заголовочных файлов) для программирования потоков. Эти рекомендации носят название POSIX threads или Pthreads.

-Создается когда функция main() (in C) или PROGRAM (in fortran) вызываеьтся загрузчиком процесса. Функция main() может создавать дочерние threads. Если основной поток завершает работу, процесс прерывается даже если внутри процесса существуют другие потоки, если только не предприняты специальные действия. Для избегания прерывания процесса можно использовать pthread\_exit().

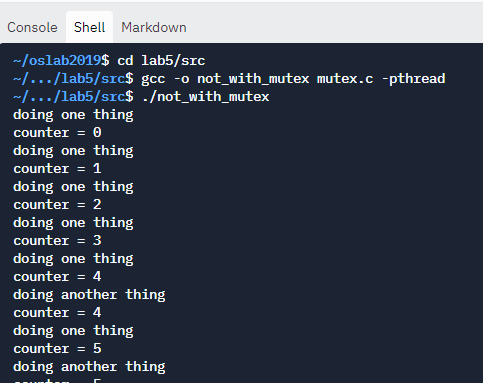
5. Как линковаться на бибилотеку pthread.

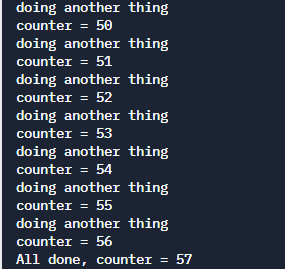
Библиотека Pthread может быть реализована на любом языке, но для соответствия стандарту POSIX она должна быть согласована со стандартизированным интерфейсом. Библиотека Pthread — не единственная реализация потокового API-интерфейса. Существуют другие реализации, созданные сторонними фирмами-производителями аппаратных и программных средств. Например, среда Sun поддерживает библиотеку Pthread и собственный вариант библиотеки потоков Solaris. В этой главе мы рассмотрим некоторые функции библиотеки Pthread, которые реализуют управление потоками.

Библиотека pthread предлагает множество функций, которые разработчик может использовать для того, чтобы создавать, уничтожать, присоединяться и синхронизировать потоки с помощью семафоров (мутексов).

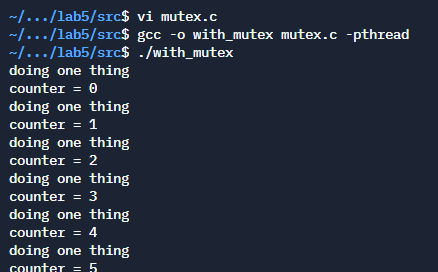
***Скомпилировать mutex.c без использования и с использованием мьютекса. Объяснить разницу в поведении программы.***

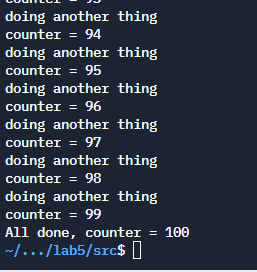
***Без использования mutex***





***Использования mutex***





***Объяснение***

Когда мьютекс не используется, thread могут обмениваться информацией через локальные переменные (common).Два потока, идущие параллельно друг другу, дают конечный результат только 57.

При использовании мьютекса для блокировки критической секции потоки по очереди переходят в критическую секцию, конечный результат только 100(50+50).

**Задание 2**

**Необходимые знания**

1.POSIX threads: как создавать, как дожидаться завершения.

-POSIX определяет набор интерфейсов (функций заголовочных файлов) для программирования потоков. Эти рекомендации носят название POSIX threads или Pthreads.

-Создается когда функция main() (in C) или PROGRAM (in fortran) вызываеьтся загрузчиком процесса. Функция main() может создавать дочерние threads. Если основной поток завершает работу, процесс прерывается даже если внутри процесса существуют другие потоки, если только не предприняты специальные действия. Для избегания прерывания процесса можно использовать pthread\_exit().

2.Как линковаться на бибилотеку pthread.

-Библиотека Pthread может быть реализована на любом языке, но для соответствия стандарту POSIX она должна быть согласована со стандартизированным интерфейсом. Библиотека Pthread — не единственная реализация потокового API-интерфейса. Существуют другие реализации, созданные сторонними фирмами-производителями аппаратных и программных средств. Например, среда Sun поддерживает библиотеку Pthread и собственный вариант библиотеки потоков Solaris. В этой главе мы рассмотрим некоторые функции библиотеки Pthread, которые реализуют управление потоками.

-Библиотека pthread предлагает множество функций, которые разработчик может использовать для того, чтобы создавать, уничтожать, присоединяться и синхронизировать потоки с помощью семафоров(мутексов).

3.Как использовать мьютексы.

-Мьютекс – это экземпляр типа pthread\_mutex\_t. Перед использованием необходимо инициализировать мьютекс функцией pthread\_mutex\_init:

int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr);

-где первый аргумент – указатель на мьютекс, а второй – аттрибуты мьютекса. Если указан NULL, то используются атрибуты по умолчанию. В случае удачной инициализации мьютекс переходит в состояние «инициализированный и свободный», а функция возвращает 0. Повторная инициализация инициализированного мьютекса приводит к неопределённому поведению.

-Если мьютекс создан статически и не имеет дополнительных параметров, то он может быть инициализирован с помощью макроса PTHREAD\_MUTEX\_INITIALIZER.

-После использования мьютекса его необходимо уничтожить с помощью функции:

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex);

-В результате функция возвращает 0 в случае успеха или может возвратить код ошибки.

-После создания мьютекса он может быть захвачен с помощью функции:

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

-После этого участок кода становится недоступным остальным потокам – их выполнение блокируется до тех пор, пока мьютекс не будет освобождён. Освобождение должен провести поток, заблокировавший мьютекс, вызовом:

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

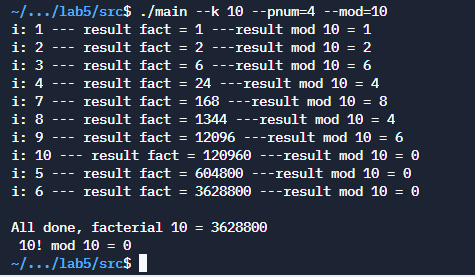
***Написать программу для паралелльного вычисления факториала по модулю mod (k!), которая будет принимать на вход следующие параметры (пример: -k 10 --pnum=4 --mod=10):***

1. ***k - число, факториал которого необходимо вычислить.***
2. ***pnum - количество потоков.***
3. ***mod - модуль факториала***

***Для синхронизации результатов необходимо использовать мьютексы.***

***(Code in github)***

https://github.com/hoaibao16/os\_lab\_2019/blob/master/lab5/src/factorial.c



**Задание 3**

**Необходимые знания**

1.Состояние deadlock.

-Взаимоблокировка (тупиковая ситуация, deadlock) — это состояние, при котором каждый поток ожидает на освобождение одного из ресурсов, а все ресурсы при этом захвачены. Потоки будут ожидать друг друга, и они никогда не смогут освободить захваченные ресурсы. Поэтому ни один из потоков не сможет продолжать выполнение, что означает наличие взаимоблокировки.

-Самый простой пример взаимоблокировки— это самоблокировка (self-deadlock). Если поток выполнения пытается захватить ту блокировку, которую он уже удерживает, то ему необходимо дождаться, пока блокировка не будет освобождена. Но поток никогда не освободит блокировку, потому что он ожидает на ее захват, и это приводит к тупиковой ситуации.

***Напишите программу для демонстрации состояния deadlock.***

(code in github)

https://github.com/hoaibao16/os\_lab\_2019/blob/master/lab5/src/deadlock.c

