Задание 1

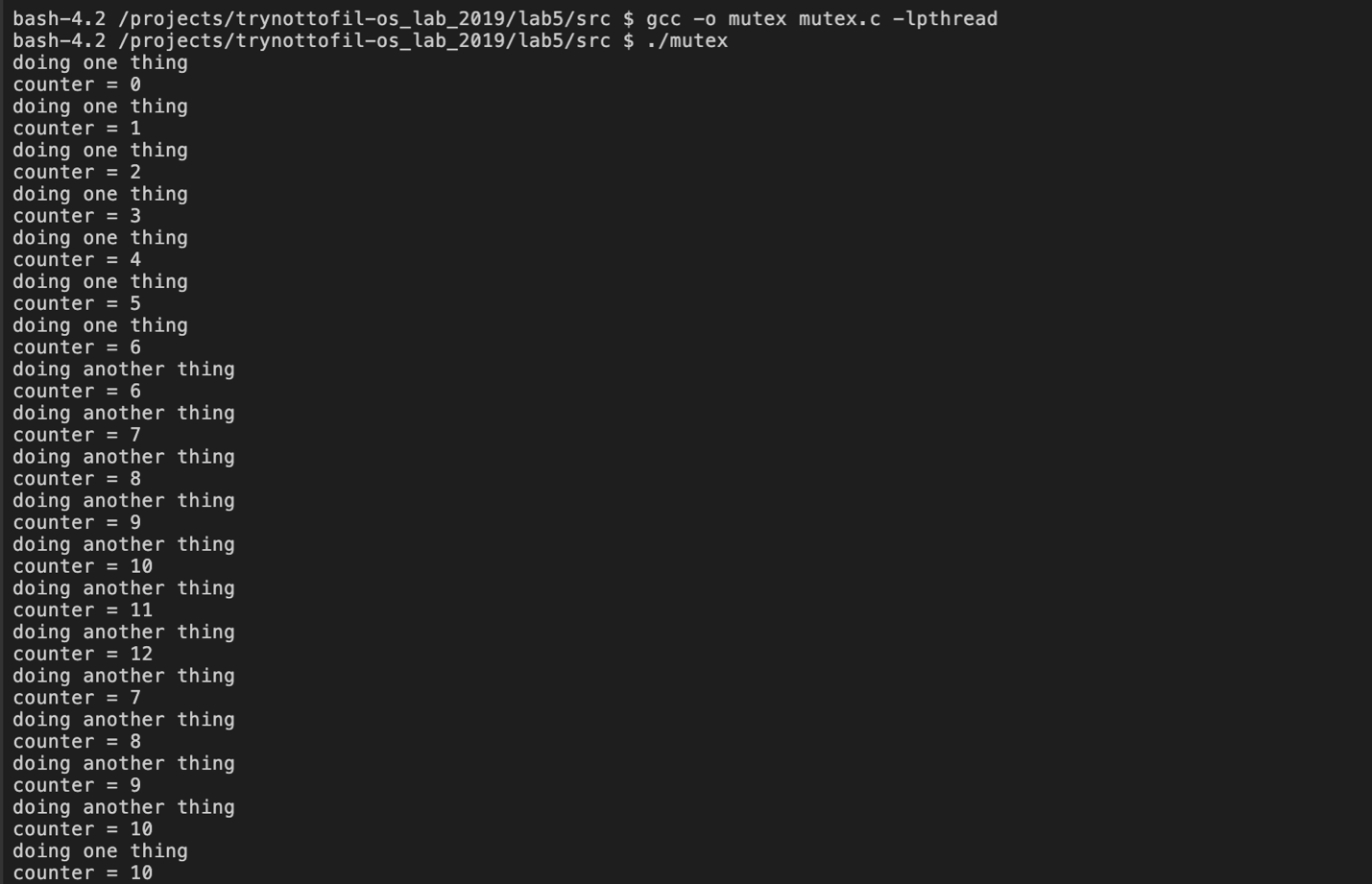
1. Компилирование программ с помощью gcc. -теория есть в предыдущих моих лабах
2. Состояние гонки.

Состояние гонки. Race condition. Другое название: гонки данных (data race). 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](http://www.viva64.com/ru/t/0048/) [многопоточного](http://www.viva64.com/ru/t/0033/) приложения пытаются одновременно получить доступ к данным, причем хотя бы один поток выполняет запись.

1. Критическая секция.

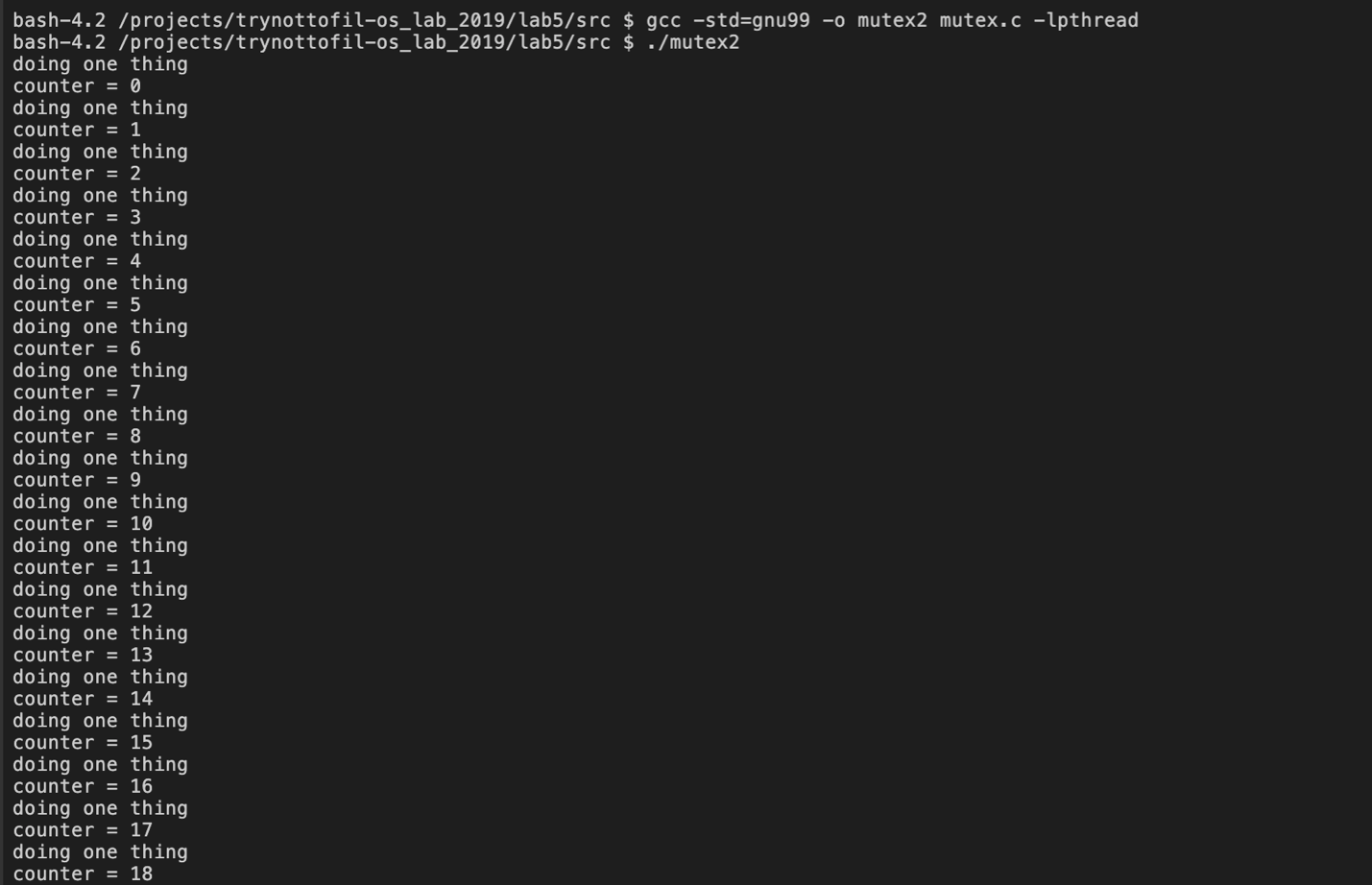
**Критическая секция** — участок исполняемого кода программы, в котором производится доступ к общему ресурсу (данным или устройству), который не должен быть одновременно использован более чем одним потоком выполнения. При нахождении в **критической секции** двух (или более) потоков возникает состояние «гонки»

1. POSIX threads: как создавать, как дожидаться завершения. -теория есть в предыдущих моих лабах
2. Как линковаться на бибилотеку pthread -теория есть в предыдущих моих лабах



……





………



Первый случай без мьютекса, а второй с. Как видно из результатов в 1 случае происходит беспорядочный доступ к общей переменной и соответственно данные выскакивают беспорядочно и повторяются. Во втором случае мьютекс включен и поэтому потоки работают по очереди и результаты упорядочены.

Чтобы мьютекс заработал его нужно было раскомментить в файле

Задание 2

1. POSIX threads: как создавать, как дожидаться завершения-теория есть в предыдущих моих лабах
2. Как линковаться на бибилотеку pthread-теория есть в предыдущих моих лабах
3. Как использовать мьютексы.

Перед использованием необходимо инициализировать мьютекс функцией **pthread\_mutex\_init**

[?](https://learnc.info/c/pthreads_mutex_introduction.html)

|  |
| --- |
| 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

[?](https://learnc.info/c/pthreads_mutex_introduction.html)

|  |
| --- |
| 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

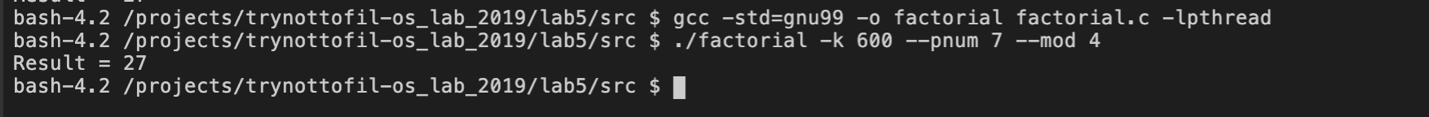
[?](https://learnc.info/c/pthreads_mutex_introduction.html)

|  |
| --- |
| 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

[?](https://learnc.info/c/pthreads_mutex_introduction.html)

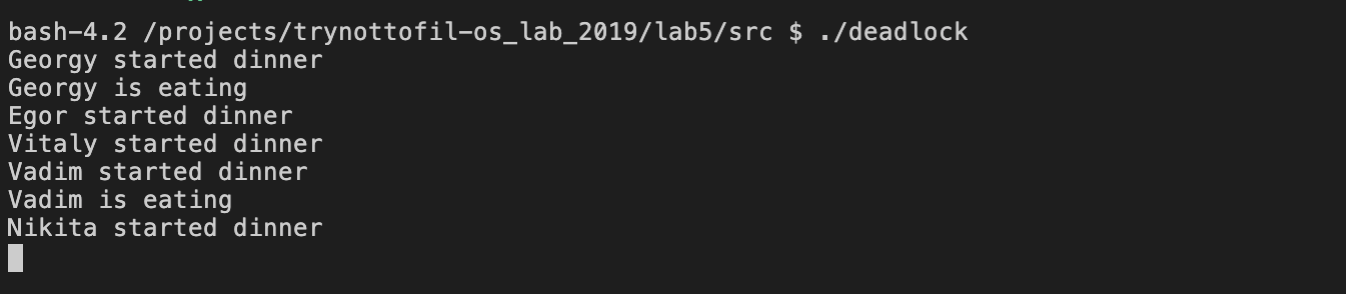
|  |
| --- |
| int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex); |



Задание 3:

Состояние deadlock:

**Deadlock** — ситуация в многозадачной среде или СУБД, при которой несколько процессов находятся в состоянии бесконечного ожидания ресурсов, захваченных самими этими процессами.



Я описал дэдлок на примере классической для него «задачи об обедающих философах»

По выводу программы видно, что мы с ребятами всё еще кушаем, вот уже целый семест, а значит столовая в дэдлоке