

2021-01-28

## Семестр 2.

### Практическое занятие #1.

#### Теоретические вопросы:

- В.1.1. Как получить идентификатор потока с помощью POSIX при помощи C и C++?
- В.1.2. В чем заключается основная проблема ввода и вывода потоков POSIX?
- В.1.3. Что такое состояние гонки, как его избежать?
- В.1.4. Что такое тупик?
- В.1.5. Что такое `std::future()` в C++ и какую проблему он пытается решить?
- В.1.6. Какова основная причина использования `std::call_once()`?
- В.1.7. В чем разница между `std::shared_mutex` и `std::mutex`?
- В.1.8. Для чего нужен рекурсивный мьютекс?
- В.1.9. Если существует возможность создавать многочисленные потоки для решения разнообразных задач в рамках одного процесса, объясните, зачем в этих условиях может понадобиться функция `fork()`.
- В.1.10. Как вы думаете, возможно ли в дочернем процессе после возврата из функции `fork()` безопасно переинициализировать переменные состояния путем их разрушения функцией `pthread_cond_destroy` и последующей инициализацией вызовом `pthread_cond_init`? Свой ответ продемонстрируйте примером.

Примечание 1: Ответ на теоретический вопрос считается полным только при условии наличия программного кода, поясняющего ответ.

#### Задачи:

- 1.1. Внесите необходимые изменения в программу `badptexit.c` так, чтобы она корректно передавала структуру данных между потоками.
- 1.2. Программа `lockrw.c` иллюстрирует применение блокировок чтения-записи. Внесите необходимые изменения в эту программу с учетом того, что должна быть предусмотрена дополнительная синхронизация, чтобы позволить главному потоку изменять идентификатор потока в задании. Какая дополнительная синхронизация должна быть предусмотрена (если она необходима), чтобы позволить главному потоку изменять идентификатор потока в задании. Как это повлияло на функцию `job_remove`?
- 1.3. Примените подход, представленный в программе `condit.c`, к программе `lockrw.c`, с которой вы работали в Задаче 1.2, для реализации функции рабочего потока. Дополните функцию `queue_init` инициализацией переменной состояния и измените функции `job_insert` и `job_append` так, чтобы они посылали сигналы рабочим потокам. Какие сложности при этом возникли?
- 1.4. Какую последовательность действий можно считать правильной и почему?
  - i. Запереть мьютекс (`pthread_mutex_lock`).
  - ii. Изменить переменную состояния, защищаемую мьютексом.
  - iii. Послать сигнал ожидающим потокам (`pthread_cond_broadcast`).

iv.Отпереть мьютекс (`pthread_mutex_unlock`).

или

v.Запереть мьютекс (`pthread_mutex_lock`).

vi.Изменить переменную состояния, защищаемую мьютексом.

vii.Отпереть мьютекс (`pthread_mutex_unlock`).

viii.Послать сигнал ожидающим потокам (`pthread_cond_broadcast`).

Свой ответ подтвердите действующим примером программы.

1.5. Какие примитивы синхронизации можно использовать для реализации барьера?

Реализуйте функцию `pthread_barrier_wait`.

Примечание: Barriers - это механизм такой синхронизации, который можно использовать для координации действий нескольких потоков, выполняющихся одновременно. Барьер позволяет каждому потоку дождаться момента, когда все сотрудничающие с ним потоки достигнут той же точки, и продолжить работу. В лекции вы уже познакомились с одной из разновидностей барьеров - функцией `pthread_join`, действующей как барьер, позволяя одному потоку дождаться завершения другого. Прототип выглядит вот так:

```
#include <pthread.h>
```

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Возвращает 0 или `PTHREAD_BARRIER_SERIAL_THREAD` в случае успеха, код ошибки - в случае неудачи.

1.6. Откомпилируйте и запустите программу из файла `p_atfork.c` в Linux, перенаправив вывод в файл. Объясните полученные результаты. Напишите набор юнит-тестов.

1.7. Если возможно, то сделайте функцию `getenv` из файла `mygetenv.c` безопасной в контексте обработки сигналов, блокируя доставку сигнала в начале функции и восстанавливая предыдущую маску сигналов перед возвратом из нее? Если считаете, что это невозможно, то объясните почему.

1.8. Создайте программу для проверки версии функции `getenv` из файла `mygetenv.c`.

Скомпилируйте и запустите вашу программу в ОС FreeBSD ( <https://www.freebsd.org/> ).

Объясните то, что получилось.

1.9. Закончите, соберите и протестируйте заготовку программы из файла `ok_sleep.c`. А потом создайте ее новую версию, в которой измените ее реализацию так, чтобы она стала безопасной в многопоточной среде, не используя функцию `nanosleep` или `clock_nanosleep`.

1.10. В файле `rmut.c` есть функция

```
void timeout(const struct timespec *when, void (*func)(void *), void *arg).
```

Попробуйте ее существенно упростить. Объясните ваше решение.

Примечание 2. Наличие юнит-тестов для написанных вами программных решений обязательно! Выбор инструментов и библиотек - на ваше усмотрение.