Конспект занятия 17.12.21

**1. Потоки.**

**Работа с потоками**

Некоторые важные функции.

**Создание потока:**

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine) (void \*), void \*arg);

**Присоединение потока:**

int pthread\_join(pthread\_t thread, void \*\*retval);

**Отсоединение потока:**

int pthread\_detach(pthread\_t thread);

Функции pthread\_join() и pthread\_detach() переводят поток в так называемое обособленное состояние. Обособленное состояние позволяет вернуть ресурсы потока обратно. Если он уже находится в обособленном состоянии, поток, вызвавший pthread\_join(), получит код ошибки EINVAL. Ранее отсоединённый поток присоединить при помощи pthread\_join() уже нельзя.

Если нам надо создать поток в рамках уже работающего процесса, нам подойдут функции pthread\_create() и clone().

У любого потока есть идентификатор. Его тип – **pthread\_t**. В частности, он используется в функциях pthread\_join() и pthread\_detach() (и много где ещё). Для того, чтобы мы могли правильно проверить, что две переменных этого типа идентифицируют один и тот же поток, нужно использовать функцию pthread\_equal(). Пример – pthread\_equal(pthr1, pthr0),

**2. dlopen(), dlsym(), dlerror(), dlclose()**. Повторите их типы, аргументы и синтаксис этих функций!

**3. Динамические библиотеки**

Если мы хотим найти динамическую библиотеку, в Linux будут проверяться следующие каталоги, переменные окружения и атрибуты:

1. Атрибут ELF-файла DT\_RPATH. Здесь по умолчанию динамические библиотеки ищутся раньше всего.
2. После этого проверяется, что перечислено в переменной окружения LD\_LIBRARY\_PATH.
3. Затем будет проверен атрибут ELF-файла DT\_RPATH.
4. После этого проверяются:
   1. Кэш /etc/ld.so.cache
   2. /lib
   3. /usr/lib

Каталог **/usr/lib** проверяется последним.

Если нам надо, чтобы некая динамическая библиотека (для простоты считаем, что она лежит в том же каталоге, что и запускаемая программа) загрузилась раньше остальных библиотек, нужно использовать **LD\_PRELOAD**.

**Имена библиотек.**

**Real name**. Это имя динамической библиотеки, в котором содержится её название, суффикс .so и хотя бы 2 цифры из её версии.

Версия динамической библиотеки состоит из четырёх составляющих:

1. Номер стадии
2. Минорный номер
3. Второй номер стадии
4. Номер релиза

Первые две составляющие являются обязательными. Последние 2 могут быть опущены.

**Soname** Это имя динамической библиотеки, в котором содержатся только её имя и суффикс .so. Никаких составных частей версии в soname нет!

**Link name**. Это имя динамической библиотеки, в котором содержится её имя, суффикс .so и ОДНА ЦИФРА из её версии. Если их больше – это уже Real name.

Если есть какие-то цифры, отделённые точками, ДО суффикса .so – это уже не имя библиотеки, а непонятно что.

**4. Завершение потоков**

Если нам надо завершить ВСЕ потоки процесса, можно использовать следующие функции:

1. exit()
2. \_exit()
3. \_EXIT()
4. Вызов return из функции main()

При вызове любой из этих функций завершается ВЕСЬ ПРОЦЕСС!

Если нам надо завершить какой-то один поток, можно использовать следующие функции:

1. Вызов return из функции нужного потока
2. pthread\_cancel()
3. pthread\_exit()

**5. Синхронизация потоков**

Предположим, что у нас есть некие данные, которые используются несколькими потоками сразу. Если они доступны этим потокам только для чтения – проблемы в этом нет (одновременно читать одни и те же данные могут разные потоки). Проблема появляется, когда несколько разных потоков должны (или могут) одновременно вносить в эти данные какие-то изменения. Чтобы не было хаоса, нужно как-то оговаривать, в каком порядке будет осуществляться доступ потоков к таким данным. Для этого используется такой механизм, как **синхронизация потоков**.

Короче говоря, синхронизация потоков нужна, когда у разных потоков есть права на запись к одним и тем же данным. Когда такие данные доступны этим потокам только для чтения – синхронизация не требуется.

Есть разные способы, при помощи которых можно синхронизировать доступ потоков к одной и той же переменной (например, типа int). С точки зрения скорости работы разумно написать перед названием типа ключевое слово **atomic**, т.е., сделать эту переменную атомарной. В частности, широко используемые мьютексы могут быть медленнее, чем операции над атомарными объектами.

Смысл атомарных объектов (любых – т.е., операций, переменных и т.д.) заключается в том, что выполняемые с ним операции можно считать неделимыми. Они не могут быть прерваны, и их результат не может быть получен до того, как такая операция закончится.

Пример **неатомарной** операции в C – инкремент. Насколько я помню, для его осуществления требуются 3 инструкции Assembler (извлечение значения из памяти, увеличение этого значения и запись обратно в память).

**6. Мьютексы. Тупики. Библиотека pthread.**

При работе с потоками может возникнуть такая ситуация, как **deadlock** (**тупик**). Если кратко, смысл в том, что два потока (или несколько потоков) взаимно блокируют друг друга. То есть, первый поток ждёт, когда освободится нужный ему ресурс, который захвачен вторым потоком. А второй поток в это же самое время ждёт, когда будет освобождён другой ресурс, который, в свою очередь, захвачен первым потоком. В результате возникает ошибка.

При работе с атомарными объектами (переменными, операциями) тупики невозможны. При этом такая ситуация возможна, когда мы используем мьютексы (тут уже многое зависит от платформы, на которой мы работаем).

**Мьютекс**, если кратко – это такой примитив синхронизации, когда в один и тот же момент времени доступ к объекту, который им захвачен, есть только у одного потока.

Пример использования мьютекса:

void\* minus(void \*args) {

int local;

//Блокировка: теперь к ресурсам имеет доступ только один поток, который владеет мьютексом. Он же единственный, кто может его разблокировать

pthread\_mutex\_lock(&mutex);

local = counter;

printf("min %d\n", counter);

local = local - 1;

counter = local;

pthread\_mutex\_unlock(&mutex);

return NULL;

}

Общее правило хорошего тона – чем меньше кода захвачено одним мьютексом, тем лучше.

Кроме мьютексов, библиотека pthread предоставляет такие примитивы синхронизации, как:

* Условные переменные (conditional variables). В Стивенсе и Раго они, если я правильно помню, называются «переменными состояния».
* Циклические блокировки (spinlocks)
* Семафоры