Потоки и управление ими

- Поддержка потоков в UNIX появилась с принятием стандарта POSIX. Согласно нему поток создается при помощи вызова:
- `#include <pthread.h>`
- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void* (*start)(void *), void *arg)`
- Упрощенно вызов `pthread_create(&thr, NULL,start,NULL)` создаст поток, который начнет выполнять функцию `start` и запишет в переменную `thr` идентификатор созданной нити.

Потоки и управление ими

- Первый аргумент этой функции `thread` - это указатель на переменную типа `pthread_t`, в которую будет записан идентификатор созданного потока, который в последствии можно будет передавать другим вызовам, когда мы захотим сделать что-либо с этим потоком. Здесь мы сталкиваемся с первой особенностью POSIX API, а именно с непрозрачностью базовых типов. Дело в том, что мы практически ничего не можем сказать про тип `pthread_t`.

Потоки и управление ими

- Единственное что сказано в стандарте, это что эти значения можно копировать, и что используя вызов `int pthread_equal(pthread_t thr1, pthread_t thr2)` мы можем установить что оба идентификатора `thr1` и `thr2` идентифицируют один и тот же поток (при этом они вполне могут быть неравны в смысле оператора равенства). Подобными свойствами обладает большинство типов используемых в данном стандарте!

Потоки и управление ими

- Второй аргумент этой функции `attr` - указатель на переменную типа `pthread_attr_t`, которая задает набор свойств создаваемой нити. Здесь мы сталкиваемся со второй особенностью POSIX API, а именно с концепцией атрибутов. Дело в том, что в этом API во всех случаях, когда при создании или инициализации некоторого объекта необходимо задать набор неких дополнительных его свойств, вместо указания этого набора при помощи набора параметров вызова используется передача предварительно сконструированного объекта, представляющего этот набор атрибутов.

Потоки и управление ими

- Такое решение имеет, по крайней мере, два преимущества. Во-первых, мы можем зафиксировать набор параметров функции без угрозы его изменения в дальнейшем, когда у этого объекта появятся новые свойства. Во-вторых, мы можем многократно использовать один и тот же набор атрибутов для создания множества объектов.

Потоки и управление ими

- Третий аргумент вызова `pthread_create` - это указатель на функцию типа `void* (*)(void *)`. Именно эту функцию и начинает выполнять вновь созданный поток, при этом в качестве параметра этой функции передается четвертый аргумент вызова `pthread_create`. Таким образом можно с одной стороны параметризовать создаваемый поток кодом, который он будет выполнять, с другой стороны параметризовать его различными данными передаваемыми коду.

Потоки и управление ими

- Функция `pthread_create` возвращает нулевое значение в случае успеха и ненулевой код ошибки в случае неудачи. Это также одна из особенностей POSIX API, вместо стандартного для Unix подхода, когда функция возвращает лишь некоторый индикатор ошибки, а код ошибки устанавливает в переменной `errno`. Функции POSIX API возвращают код ошибки в результате своего аргумента.

Потоки и управление ими

- Очевидно, это связано с тем, что с появлением в программе нескольких потоков, вызывающих различные функции и возвращающие код ошибки в одну и ту же глобальную переменную `errno`, наступает полная неразбериха. И хотя из-за огромного числа функций, уже использующих `errno`, библиотека потоков и обеспечивает по экземпляру `errno` для каждого потока, однако создатели стандарта выбрали более правильный, а главное - более быстрый подход, при котором функции API просто возвращают коды ошибки.

Потоки и управление ими

- В качестве резюме рассмотрим пример.
- Заметим, что хотя функции работы с потоками описаны в файле включения `pthread.h`, на самом деле они находятся в библиотеке. Поэтому процесс компиляции и сборки многопоточной программы выполняется в два этапа:
- `gcc -Wall -c -o test.o test.c`
- `gcc -Wall -o test test.o <path>libgcc.a -lpthread`
- В большинстве версий Linux библиотека лежит в `/usr/lib/`.

Завершение потока

- Поток завершается, когда происходит возврат из функции. Если мы хотим получить возвращаемое этой функцией значение, то мы должны воспользоваться такой функцией:
- `int pthread_join(pthread_t thread, void** value_ptr)`
- Эта функция дожидается завершения потока с идентификатором `thread`, и записывает возвращаемое ею значение в переменную, на которую указывает `value_ptr`. При этом освобождаются все ресурсы связанные с потоком, потому эта функция может быть вызвана для данного потока только один раз.

Завершение потока

- Если нас чем-то не устраивает возврат значения через `pthread_join`, например, необходимо получить данные из нескольких потоков, то следует воспользоваться каким либо другим механизмом, например, организовать очередь возвращаемых значений, или возвращать значение в структуре, указатель на которую передают в качестве параметра потока. То есть использование `pthread_join` - это вопрос удобства, а не догма, в отличие от случая пары `fork() - wait()`.

Завершение потока

- В случае, если мы хотим использовать другой механизм возврата или нас просто не интересует возвращаемое значение, то мы можем отсоединить (*detach*) поток, сказав тем самым, что мы хотим освободить ресурсы, связанные с потоком, сразу по завершению функции потока. Сделать это можно несколькими способами. Во-первых, можно сразу создать поток отсоединенным, задав соответствующий объект атрибутов при вызове `pthread_create`.

Завершение потока

- Во-вторых, любой поток можно отсоединить, вызвав в любой момент его жизни (то есть до вызова `pthread_join()`) функцию `int pthread_detach(pthread_t thread)`, и указав ей в качестве параметра идентификатор потока. При этом поток вполне может отсоединить сам себя, получив свой идентификатор при помощи функции `pthread_t pthread_self(void)`. Следует подчеркнуть, что отсоединение потока никоим образом не влияет на процесс его выполнения, а просто помечает поток как готовый по своему завершению к освобождению ресурсов.

Завершение потока

- Под освобождаемыми ресурсами подразумеваются в первую очередь стек, память, в которую сохраняется контекст потока, данные, специфичные для потока и тому подобное. Сюда не входят ресурсы выделяемые явно, например, память, выделяемая через `malloc`, или открываемые файлы. Подобные ресурсы следует освобождать явно и ответственность за это лежит на программисте.

Завершение потока

- Помимо возврата из функции потока существует вызов, аналогичный вызову `exit()` для процессов:
- `int pthread_exit(void *value_ptr)`
- Этот вызов завершает выполняемый поток, возвращая в качестве результата его выполнения `value_ptr`. Реально при вызове этой функции поток из нее просто не возвращается. Помните, что функция `exit()` по-прежнему завершает процесс, то есть в том числе уничтожает все потоки.

Завершение потока

- Рассмотрим соответствующий пример.
- Заметим, что способ обработки запроса на прерывание потока зависит от состояния указанного потока. Две функции, `pthread_setcancelstate()` и `pthread_setcanceltype()`, определяют это состояние.

Особенности главного потока

- Как известно, программа на Си начинается с выполнения функции `main()`. Поток, в котором выполняется данная функция, называется главным или начальным (так как это первый поток в приложении). С одной стороны этот поток обладает многими свойствами обычного потока, для него можно получить идентификатор, он может быть отсоединен, для него можно вызвать `pthread_join` из какого-либо другого потока. С другой стороны он обладает некоторыми особенностями, отличающих его от других потоков.

Особенности главного потока

- Возврат из этого потока завершает весь процесс, что бывает иногда удобно, так как не надо явно заботиться о завершении остальных потоков. Если мы не хотим, чтобы по завершении этого потока остальные потоки были уничтожены, то следует воспользоваться функцией `pthread_exit`.
- У функции этого потока не один параметр типа `void*`, как у остальных, а пара `argc-argv`. Строго говоря функция `main` не является функцией потока, так как в большинстве ОС она сама вызывается некими функциями, которые подготавливают ее выполнение.

Особенности главного потока

- В-третьих, многие реализации отводят на стек начального потока гораздо больше памяти, чем на стеки остальных потоков. Очевидно, это связано с тем что существует много однопоточных приложений (то есть традиционных приложений), требующих значительного объема стека, а от автора нового многопоточного приложения можно потребовать ограниченности аппетитов.

Жизненный цикл потока

- Рассмотрим жизненный цикл потока, а именно последовательность состояний, в которых пребывает поток за время своего существования. В целом можно выделить четыре таких состояния:

Жизненный цикл потока

Состояние потока	Что означает
Готов /Ready/	Поток готов к выполнению, но ожидает процессора. Возможно он только что был создан, был вытеснен с процессора другим потоком, или только что был разблокирован (вышел из соответствующего состояния).

Жизненный цикл потока

Состояние потока	Что означает
Выполняется /Running/	Поток сейчас выполняется. Следует заметить, что на многопроцессорной машине может быть несколько потоков в таком состоянии.
Заблокирован /Blocked/	Поток не может выполняться, так как ожидает чего-либо. Например, окончания операции ввода-вывода.

Жизненный цикл потока

Состояние потока	Что означает
Завершен /Terminated/	Поток была завершен, например, вследствие возврата из функции потока, вызова `pthread_exit`. Поток не был отсоединен и для него не была вызвана функция `pthread_join`. Как только происходит одно из этих событий, поток перестает существовать.

Жизненный цикл потока

- Потоки могут создаваться системой, например, начальный поток, который создается при создании процесса, или при помощи явных вызовов `pthread_create()` пользовательским процессом. Однако любой создаваемый поток начинает свою жизнь в состоянии "готов". После чего в зависимости от политики планирования системы он может либо сразу перейти в состояние "выполняется", либо перейти в него через некоторое время.

Жизненный цикл потока

- Необходимо обратить внимание на типичную ошибку, совершаемую многими. Она заключается в том, что в отсутствии явных мер по синхронизации потоков предполагается, что после возврата из функции `pthread_create` новый поток будет существовать. Однако это не так, ибо при определенной политике планирования и атрибутах потока вполне может статься, что новый поток уже успеет выполниться к моменту возврата из этой функции.