## ТЕМА: "Параллельная обработка нескольких клиентов"

Цель лабораторной работы: ознакомиться c принципами разработки и функционирования серверных приложений, использующих сокеты и способных обрабатывать нескольких клиентов одновременно.

### **Ход работы**

1. **Параллельная обработка с помощью порождения дочернего процесса под каждый подключающийся клиент**

Измените серверное приложение из лабораторной работы "Основы использования сокетов" так, чтобы оно было способно обрабатывать нескольких клиентов одновременно. Для параллельной (одновременной) обработки нескольких клиентов в приложении должен быть реализован механизм на основе порождения дочерних процессов при помощи системного вызова **fork**.

Основной процесс занимается только приемом входящих соединений и порождением процессов под обработку запросов подключающихся клиентов. Диалог с клиентом ведет соответствующий дочерний процесс.

* 1. Как только сервер принимает входящее соединение от нового клиента, он порождает новый процесс – копию самого себя с помощью функции fork.
  2. Родительский процесс закрывает сокет для обмена данными с новым клиентом (за его ненадобностью, т.к. у дочернего процесса будет своя копия сокета) и возвращается в состояние ожидания входящего соединения.
  3. Дочерний процесс закрывает сокет для прослушки (приема входящих соединений – аналогично за его ненадобностью). Затем дочерний процесс ведет диалог (обмен данными) с клиентом. По завершении работы с клиентом дочерний процесс закрывает логическое соединение с ним, закрывая сокет, а затем завершается.

1. **Параллельная обработка с помощью порождения потока под каждый подключающийся клиент**

Измените серверное приложение из лабораторной работы "Основы использования сокетов" так, чтобы оно было способно обрабатывать нескольких клиентов одновременно. Для параллельной (одновременной) обработки нескольких клиентов в приложении должен быть реализован механизм на основе порождения отсоединенных потоков при помощи pthread\_create и pthread\_detach.

Алгоритм, в целом, аналогичен предыдущему. Вместо процессов создаются более легковесные структуры – потоки.

Основной поток блокируется в вызове функции accept, и каждый раз, когда прибывает новое клиентское соединение, функцией pthread\_create создается новый поток. Функция, выполняемая новым потоком, — это отдельная функция, ее аргументом является присоединенный сокет. Функция выполняется как отсоединенный (detached) поток, потому что основному потоку не требуется ждать ее завершения. Функция, выполняемая новым потоком, вызывает расчетную функцию. Когда эта функция возвращает управление, присоединенный сокет закрывается.

Эта простая версия с использованием потоков является более быстродействующей, чем даже самая быстрая из версий с предварительным порождением процессов. Кроме того, эта версия, в которой каждый клиент обслуживается одним потоком, во много раз быстрее версии, в которой каждый клиент обслуживается специально созданным для него дочерним процессом.

<https://it.wikireading.ru/7513>

Теория - Книга Стивенса, Раго UNIX: Профессиональное программирование

### **Теоретическая часть**

**Порождение процессов**

Параллельная обработка клиентов сервером на основе порождения дочерних процессов подразумевает создание дочернего процесса для обслуживания каждого нового клиента, тогда как родительский процесс занимается только прослушиванием порта и приёмом соединений.

Чтобы добиться такого поведения, сразу после **accept** сервер вызывает функцию **fork** для создания дочернего процесса. Далее анализируется значение, которое вернула эта функция. В родительском процессе оно содержит идентификатор дочернего, а в дочернем процессе равно нулю. Используя этот признак, мы переходим к очередному вызову **accept** в родительском процессе, а дочерний процесс обслуживает клиента и завершается (**\_exit**).

Функция **fork** имеет следующий синтаксис:

#include <unistd.h>

pid\_t fork(void);

Новый процесс, созданный функцией **fork**, называется дочерним процессом, или процессом-потомком. Эта функция вызывается один раз, а управление возвра­щает дважды, с единственным отличием: в дочернем процессе она возвращает 0, а в родительском — идентификатор созданного дочернего процесса. Последнее обстоятельство объясняется тем, что процесс может иметь несколько потомков, а система не предусматривает функций, с помощью которых можно было бы по­лучить идентификаторы дочерних процессов. В дочернем процессе функция **fork** возвращает 0, поскольку дочерний процесс имеет только одного родителя и всегда может получить его идентификатор с помощью функции **getppid**. (Идентифика­тор процесса 0 зарезервирован за ядром, поэтому невозможно получить 0 в каче­стве идентификатора дочернего процесса.)

И родительский и дочерний процессы продолжают выполнение программы с ин­струкции, следующей за вызовом функции **fork**. Процесс-потомок является точ­ной копией родительского процесса. Например, потомок получает копии сегмента данных, кучи и стека родителя; роди­тельский и дочерний процессы не используют совместно одни и те же области памяти. Но они совместно используют сегмент кода.

**Порождение потоков**

**TBA**