Крищенко В. А.

# Основы программирования в POSIX-системах

Учебное пособие. Черновик от 18 октября 2012 г.

## Реферат

В пособии рассмотрены основные концепции POSIX-систем и предоставляемые ими разработчику программ на язык Си функции. Изложение ведется в рамках разработки примера — сервиса хранения в памяти данных вида «ключ-значение», предоставляющим клиентам доступ к этим данным через локальные сокеты.

Пособие предназначено для студентов специальностей, связанных с программированием.

## Содержание

# Введение

Пособие предполагает хорошее понимание языка Си.

Стандарты POSIX не затрагивают вопросы графического интерфейса пользователя (вопросы стандартизации части связанных с этим вопросов занимается freedesktop и X11, но основные прикладные API не стандартизированы).

Подобное справочное руководство [**?**].

# 1 Обзор примеров

## 1.1 Выбор языка программирования

Естественным выбором для написания примеров программ для POSIX-систем является язык Си: все системные функции POSIX выглядят для программиста как функции на языке Си, а компилятор языка Си должен быть в любой отвечающей стандарту POSIX-системе[1]. Более того, иных штатных компиляторов стандарт POSIX не требует.

Для управления процессом компиляции и сборки программ-примеров...

## 1.2 Организация исходного кода

При написании примеров автору хотелось, что бы при проработке каждой темы читатель мог ограничится рассмотрением ограниченного объёма кода, который бы имел ясное предназначение и транслировался бы в библиотеку или исполняемый файл. По этой причине была выбрана следующая организация исходных текстов примеров: весь исходный код разбит на большое число небольших связанных друг с другом подпроектов.

Каждый проект находятся в своем каталоге. Заголовочные файлы проекта хранятся в подкаталоге **inc**, исходные файлы — в подкаталоге **src**. Каждый проект содержит сценарий сборки цели построения и файл с функциями-тестами. Функция **main** исполняемого файла находится в файле **src/main.c**. Функции для тестирования находятся в файле **tests/run.c**.

Большинство проектов относится к одному из следующих видов:

1) проект, содержащий только заголовочные файлы с макросами и inline-функциями (и тесты для них);
2) проект для создания статической библиотеки;
3) проект для создания динамической библиотеки;
4) проект для создания исполняемого файла.

## 1.3 Декомпозиция программы-примера

На протяжении пособия будет рассмотрено создание сервиса для хранения в памяти данных вида «ключ-значение».

В пособии при проектировании примеров использовался комбинированный «встречный» подход. На верхнем уровне разрабатываемая программная система была декомпозирована «сверху вниз» и были выделены следующие крупные составляющие (рис. 1.1):

1) библиотеки для журналирования вывода в **stdout** и **stderr**;
2) библиотеки для хранения ассоциативных данных;
3) серверы хранения данных, обслуживающие локальных клиентов;
4) программа для запуска служб.

Затем большая часть кода была спроектирована «снизу вверх», начиная со вспомогательных макросов и функций.

[1] **c99**: `http://pubs.opengroup.org/onlinepubs/9699919799/utilities/c99.html`

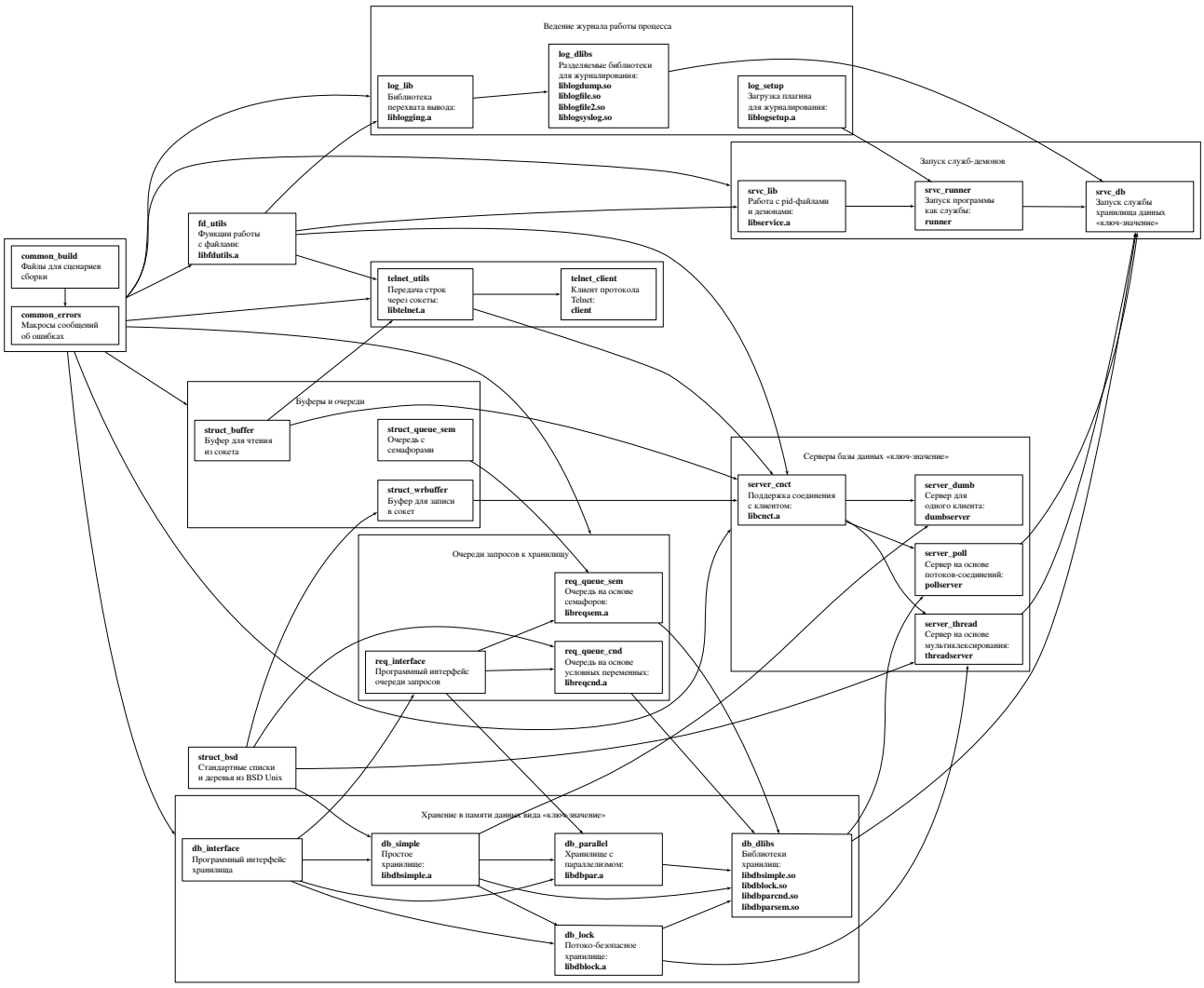


Рисунок 1.1 — Взаимосвязи и цели построения проектов-примеров

# 2 Операционные системы семейства POSIX

## 2.1 Стандарты группы POSIX

К концу 80-х годов в мире существовало несколько систем-наследников Unix System V, а также системы семейства BSD. Хотя основные их концепции совпадали, как и значительное подмножество их системных вызовов, количество различий между ними росло от версии к версии. Это обстоятельство мешало тем разработчикам программного обеспечения, которые стреимились создавать переносимые на уровне исходных текстов программы и приводило к... После событий, известных как «войны unix» и сокращения доли UNIX-систем под напором Windows NT, был создан консорциум для стандартизации unix-подобных операционных систем под названием Open Group.

Поскольку название UNIX является торговой маркой, в качестве названия для создаваемых стандартов было выбрано (кем?) название POSIX. Основным действующим на момент написания пособия стандартом POSIX является стандарт POSIX 2008.1, который доступен в электронном виде[1].

[1] `http://pubs.opengroup.org/onlinepubs/9699919799/`

### 2.1.1 Цели

Группа стандартов POSIX прежде всего создана для переносимости программ на уровне исходных текстов. Переносимость программ и библиотек в машинном коде она не рассматривает. На практике даже в случае одинаковых центральных процессоров программы в бинарном виде не являются переносимыми уже в силу разных форматов исполняемых файлов.

## 2.2 Основные POSIX-совместимые системы

Большая часть стандартов POSIX была принята пост-фактум уже после появления некоторых возможностей в той или иной операционной системе. Негативным следствем этого является то, что консистетность стандартных системных интерфейсов не слишком высока: разные интерфейсы используют разные стили именования и чуть различающиеся способы уведомления об ошибке. Позитивным следствием появления реализации до самого стандарта является его подтверденность практикой.

Изначально стандарты POSIX были попыткой договора между крупными поставщиками UNIX-систем, такими как IBM (AIX), HP (HP-UX), Sun (Solaris) при участии разработчиков BSD Unix.. В настоящий момент доля всех этих систем сократилась до небольшой, и лишь две последних активно развиваются, а наиболее распространёнными POSIX-системами являются дистрибутивы GNU/Linux (Ubuntu, Debian, Redhat и др.) и OS X компании Apple. Дистрибутив Debian при этом в настоящий момент может использовать как ядро Linux, так и ядро от систем FreeBSD. Далее мы будем называть такие системы типичными POSIX-системами. Ссылки на них нам понадобятся, когда захочется рассмотреть не только интерфейс POSIX, но и коснуться ключевых моментов его реализации. Однако, мы не будем углубляться в реализацию. Изучить внутреннее устройство ядер операционных систем и физического устройства можно по следующим изданиям:

1)

2)

## 2.3 Общие свойства POSIX-систем

Соответствующие стандарту POSIX-системы должны поддерживать виртуальную память, а каждый процесс должен иметь своё собственное виртуальное адресное пространство. Это следует из того, что для создания нового процесса в POSIX-системах используется функция **fork()**, которая создаёт новую самостоятельную копию текущего процесса. Рассмотрим следующий пример (с целью упрощения обработка ошибок в нём отсутствует).

Листинг 2.1 — Пример создания дочернего процесса (файл **fork.c**)

```
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    int a = 0;
    int *p = &a;
    if (fork() == 0) {
        a = 10;
        sleep(1);
        printf("Child process:  p = %p, *p = %d\n", p, *p);
    } else {
        a = 20;
        sleep(1);
        printf("Parent process: p = %p, *p = %d\n", p, *p);
    }
```

```
}
```

После возврата из функции **fork()** каждый процесс имеет свою собственную переменную **a**, но значение адреса в указателе **p** у них будет одинаково. Следовательно, в переменной **p** хранится виртуальный адрес, а не физический.

Спецификация POSIX ничего не говорит об организации виртуальной памяти. Типичный POSIX-системы поддерживают страничную виртуальную память[1] с поддержкой выгрузки измененых страниц в область подкачки на диске. Возможность выгрузки памяти на диск также не требуется стандартом POSIX.

## 2.4 Терминология

Для краткости мы будем называть прикладным программистом разработчика, который пишет программое обеспечение, использующее системный интерфейс POSIX (системный программист, очевидно, реализует данный интерфейс). Программы, использующие такие системные интерфейсы, мы будем называть прикладными.

### 2.4.1 Системные функции

Стандартый POSIX не делают различий между функциями, реализованными в стандартной библиотеке Си, и функциями, которые реализоваными системным вызовом ядра ОС. И те, и другие рассматриваются как часть системного интерфейса (*system interface*) вместе с константами, типами и макросами. В пособии будет использоваться далее термин «системная функция».

Отделение системных вызовов от системных функций позволяет добиться переносимости программ между POSIX-системами с различающимися интерфейсами ядра операционной системы. Например, системная функция **fork()** в системе FreeBSD приводит к одноимённому системному вызову, а а системах GNU/Linux эта же системная функция приводит к специфичному для ядра Linux системному вызову **clone** с некоторым набором параметров. Отметим, что мнемонические имена системных вызовов, как известно, условны, поскольку пользовательская часть процесса сообщает ядру его номер, а не имя, причем номер зависит от конкретного ядра ОС. Например, номер системного вызова **fork** в ядре FreeBSD — NN, номер аналогичного (но уже не используемого) вызова в ядре Linux — MM, а номер вызова **clone** — KK. Мнемонические имена системных вызовов используются просто для удобства ссылок для них.

Для выяснения связи между системными функциями и системными вызовами ядра ОС проще всего воспользоваться программой трасссировки системных вызовов. Это strace и др.

### 2.4.2 Блокирование

Исполнение потока выполнения (задачи?) может быть приостановлено ядром ОС по двум основным причинам:

1) переход в состояние готовности: у задачи «отняли» логический процессор[2], поскольку он нужен другим процессам;

2) переход в состояние ожидания: задача снят с процессора, поскольку до наступления некоторого события он в нём не нуждается.

В состоянии готовности задача может продолжить своё выполнение в любой момент, когда планировщик задач выделит ему процессор. В состоянии блокировки задача не может

[1] Сегметная составляющая преобразования виртуального адреса в физический, имеющаяся у процессоров x86, в типичных ОС настрорено как вырожденное.

[2] По причине широкого распространения многоядерных центральных процессоров, под логическим процессором понимается нечто, способное одновременно выполнять единственный поток команд.

продолжить свое выполнение, при наступлении ожидаемого события он переводится в состояние готовности.

События, ожидаемые задачей, можно разделить на две группы: прозрачные для разработчика прикладных программ, и непрозрачные. Первые из них ожидают событий, инициированных в ядре ОС, вторые — событий, являющихся прямым следствием действий прикладных программ. Например, вызов системной функции **read()** для чтения из файла может привести к ожиданию завершения операции ввода-вывода с дисковым накопителем, но с точки зрения прикладного программиста (кто это?) эта смена состояний будет незаметна. Вызов системной функции ожидания семафора **sem_wait()** приводит к ожиданию открытия семафора другой задачей. С точки зрения планировщика задач оба этих ожидание очень похожи, различается лишь ожидаемое событие. Для прикладного программиста, наоборот, ожидание завершения ввода-вывода прозрачно, а ожидание семафора требует его открытия другой задачей. В силу этого у системных функций, ожидающих непрозрачных собйтий, обычно есть вариант с ограничением времени ожидания, например, **sem_trywait()**.

В дальнейшем мы будем называть *блокировкой* перевод процесса в состояние ожидание событий, явно зависящих от выполнения прикладных программ. Поскольку стандарты POSIX не рассматривают какие-либо концепции ядра операционной системы, то в них данный термин используется в таком же ограниченном прикладном значении.

Блокировка может привезти к неограниченному ожиданию или к взаимной блокировке (*тупику*) в случае некорректной прикладной программы. Один и тот же системный вызов может приводить и к блокировке, и к прозрачному ожиданию. Например, системная функция чтения **read()** приводит к блокировке в случае чтения из канала, связыющего два процесса, поскольку ожидает события от другой прикладной задачи, а именно записи в канал.

### 2.5 Построение программ

Для сборки программ в POSIX-системах традиционно испольуются сценарии сборки в виде программы Make, называемых обычно make-файлов. Такие файлы используют логику предикатов для описания зависимостей, и язык сценариев shell для описания команд для построения.

К сожалению, между разными POSIX-системами существуют различия, мешающих написание универсальных сценариев сборки вручную. Например, функция для ручного использования разделяемых библиотек (**dlopen** и др.) в GNU/Linux системах находятся в отдельной библиотеке **libdl.so** и требуют поэтому указания соответствующего ключа (**-ldl**) при компоновке исполняемого файла. В BSD-системах эти функции включены в основной файл стандартной библиотеки Си **libc.so**, отдельной библиотеки для них нет, и поэтому ключ **-ldl** вызовет ошибку. Оба варианта не противоречат стандарту POSIX, который не регламентирует размещение системных функий по разделяемым библиотекам. Кроме того, в разных POSIX-системах установлены разные варианты программы Make, испольщие несколько различные языки сценариев сборки[1].

В силу такого рода различий, а так же желания проверять наличие библиотек конкретных версий, для сколь-нибудь сложных проектов вместо написания сценариев сборки используют ПО для генерации сценариев под конкретное сборочное окружение, такие как Autotools, Cmake и др. Для приведенных в книге пособий мы, тем не менее, ограничимся использованием сценариев сборки для программы GNU Make[2]. В BSD-системах он обычно присутствует под именем **gmake**, хотя и не установлен по умолчанию.

Вся «чёрная магия» сценариев сборки вынесена в файл **std.mk** в проекте **common_build**. Этот файл будет включаться в сценарии сборки всех остальных проектов пособия.

[1] В GNU системах это GNU Make, в BSD — BSD Make, а в Windows — Microsoft NMake.

[2] Документация ПО GNU make: .

Поскольку навыки написания сценариев Make являются навыком, достаточно ортогональным теме пособия, то мы не будем вдаваться детали, процесса сборки, отметив лишь основные моменты. Благодаря правилам, содержащимся в файле **std.mk**, сборку можно описать декларативно, задав значения нескольким переменным значения имен некоторых файлов и каталогов. Перед включением в сценарий сборки файла **std.mk** нужно присвоить значения следующим переменным сценария (имена нескольких файлов при этом разделяются пробелами, любая переменная необязательна).

1) Переменная **INCD**: каталоги с заголовочными файлами.
2) Переменная **TARGETLIB**: создавамой статической библиотеки.
3) Переменная **TARGETEXEC**: создаваемый исполняемый файл.
4) Переменная **SRC**: компилируемые исходные файлы (кроме **src/main.c**).
5) Переменная **LIBS**: необходимые исполняемому файлу статические библиотеки.

Для проведения тестирования используются две переменнные.

1) Переменная **TESTLIBS**: необходимые для тестирования статические библиотеки (если не указано, то будет равно значению переменной **LIBS**).

2) Переменная **TESTINCD**: необходимые для тестирования каталоги с заголовочными файлами (если не указано, то будет равно значению переменной **INCD**).

При указании имени библиотеки в переменной **TESTLIBS** или **LIBS** в соответствующую переменную (**TESTINCD** или **INCD**) автоматически добавляется подкаталог **inc** каталога бибилотеки.

### 2.6 Использованные платформы

Таблица 2.1 — Протестированные платформы

| № | ОС | Ядро ОС | ЦП | Компилятор |
|---|---|---|---|---|
| 1 | FreeBSD 9.0 | | amd64 | GCC 4.2 |
| 2 | | | | CLang 3.0 |
| 3 | Debian 6.0 | Linux 3.2 | | GCC 4.7 |
| 4 | | Linux 2.6.32 | | GCC 4.4 |
| 5 | | | x86 | |
| 6 | | kFreeBSD 8.1 | | |
| 7 | OpenBSD 5.1 | | | GCC 4.2 |
| 8 | | | | CLang 3.0 |
| 9 | Ubuntu 12.04 | Linux 3.0 | | |
| 10 | | | | GCC 4.6 |
| 11 | Ubuntu 11.10 | Linux 2.6.39 | arm-8 | |
| 12 | Maemo 5.0 | Linux 2.6.28 | | GCC 4.2 |

## 3 Обработка ошибок

При разработке на языке Си программ, использующих системные функции POSIX-системы, первостепенным вопросом является корректная обработка ошибок, возникающих при вызове таких функций. Подавляющее большинство число системных функций может вернуть результат, свидетельствующий об ошибке при её выполнении. Правильная обработка таких результатов в случае разработки ПО на языке Си полностью ложиться на программиста.

### 3.1 Ошибки времени исполнения

Ошибки времени исполнения, конечно, не ограничиваются обнаруживаемыми системными функциями. Любая функция нашей программы может проверять логическую корректность её аргументов. Проверяемые условия формулируются виде утверждений ((*asserts*)). Утверждения используются как элемент защитного программирования и для диагностики корректности структур данных в тех случаях, когда это уместно.

Такие ошибки, свидетельствующие о неверной работе самой программы, мы будем далее будем называть внутренними. Их следует отделять от внешних ошибок, возникающие в силу невозможность выполнить системную функцию по причинам, часто независящим от программы.

Системные функции обычно обнаруживают внутренние ошибки, но ряд системных функций также обнаруживают внутренние ошибки, например, системная функция может сообщить о принципиально недопустимых значениях переданных ей параметров. Такие ошибки родственный проверке утверждений, поскольку также свидетельствует о принципиально неверной работе программы, а в корректно работающей программе они не должны возникнуть. Их возникновение свидетельствует об ошибке в программном коде, приведшей к потере логической целостности значений переменных, возможно, даже из-за порчи памяти процесса.

В некоторых случаях ошибку нельзя гарантировано отнести к внутренней или внешней (например, ошибка открытия файла из-за его отсутствия), в этом случае их следует считать внешними. Все заведомо внутренние ошибки можно рассматривать как фатальные: если у нас наблюдается порча значений переменных, то мы не можем гарантировать дальнейшую корректную работу программы, и её выполнение лучше аварийно прервать.

### 3.2 Корректная обработка ошибок

Корректная обработка ошибок системных функций POSIX в пределах одной функции должна включать:

1) правильная проверка результата вызываемой функции на признак ошибки;
2) добавления сообщения об ошибке в журнал работы (стек вызовов в posix?);
3) освобождение всех выделенных ресурсов, которые становятся ненужными в силу произошедшей ошибки;
4) в случае внешней ошибки — сообщение об ошибке «наверх» в место вызова функции;
5) в случае внутренней ошибки — (возможно) прекращение работы процесса.

Основной принцип обработки внешних ошибок — сообщайте об ошибках «наверх», не забывая корректно освободить занятые ресурсы и выводя сообщения об ошибке в поток **stderr**. «Наверху» разберутся, завершать ли в случае ошибки работу программы, или продолжить её выполнение.

Как минимум в двух случаях это правило в примерах не используется. Первый случай — при вызове системной функции происходит ошибка, говорящая об ошибке в программе. Например, функция поднятия семафора **sem_post()** может завершится только с ошибкой, сообщающей, что ей передали указатель не на дескриптор семафора, или превышено максимальное значение семафора.

Обе эти ошибки говорят о том, что программа работает некорректно и, возможно, её память «испорчена». По мнению автора, здесь допустимо завершить работу процесса немедленно.

Второй случай — проверка результатов функций в функциях-тестах. Если системная или тестирующая функция вернула признак внешней ошибки, то обычно тест можно завершить аварийно, причём просто используя макрос **assert**, благо компиляция тестов с заданным **NDEBUG** явно бессмысленна (рис 3.1).

Листинг 3.1 — Проверка ошибок в тестах

```
/// Sets testing suit up.
int init(void)
{
    char *template = "/tmp/test_socket_XXXXXX";
    strcpy(socket_name, template);
    int tmp_fd = mkstemp(socket_name);
    assert(tmp_fd != -1);
    assert(close(tmp_fd) != -1);
    server_pid = fork();
    assert(server_pid != -1);
    if (server_pid == 0) {
        struct simple_db simple_db;
        struct vdb *db = init_simple_db(&simple_db, NULL);
        assert(db);
        int binded_socket = create_binded_socket(socket_name);
        assert(binded_socket != -1);
        run_server(binded_socket, db);
        db->destroy(db);
        exit(EXIT_SUCCESS);
    }
    return 0;
}
```

Очевидным исключением являются тесты, созданные как раз для проверки поведения при внешних ошибках.

### 3.3 Реализация проверки утверждений

Для проверки утверждений принято использовать не условный рператор, а стандартный макрос **assert()**. Если его аргумент вычисляется в ложное значение (то есть, имеет значение 0), то выполнение программы аварийно завершается.

Поскольку **assert()** является макросом, то являющееся его аргументом выражение не должно иметь побочных эффектов. Кроме того, при определении макроса **NDEBUG** все макросы **assert()** будут раскрыты в пустые выражения[1], и их аргументы исчезнут. Поскольку компиляция с определением макроса **NDEBUG** вполне возможна в случае готовой к использованию версии программы, то конструкции типа показанных в листинге 3.2 нельзя использовать в коде, кроме кода тестирующих функций.

Листинг 3.2 — Неверное использование макроса **assert()**

```
// Is this function is always parallel-safe?
void psafe_intsort(pthread_mutex_t *m, int *a, int n)
{
    assert(!pthread_mutex_lock(m)); // This is a BAD idea.
    qsort(a, n, sizeof(int), int_compare);
    assert(!pthread_mutex_unlock(m));
}
```

Тут надо написать о пользе макроса **NDEBUG** в промышленном коде!

Приведенная в листинге 3.2 функция перестанет быть потоко-безопасной в случае компиляции с параметром **-D NDEBUG**. Поскольку в какой-то момент слишком борящийся за производительность сборщик вполне может скомпилировать вполне отлаженный исходный кол с таким параметром, желая избавиться от ухудшающих производительность проверок, к такому повороту событий следует быть готовым и не использовать проверку выражений с побочными эффектами. Кроме того, в указанном случае не будет выведена дополнительная информация о конкретном кода ошибки, которую можно было бы получить с помощью функции **strerror()**.

[1] Точнее, во что-то типа **((void) 0)**.

В силу приведенных выше соображений, в коде самих примеров макрос **assert()** используется только для диагностики, то есть прверки утверждений, которые должны быть всегда верны в корректно работающей программе, вне зависимости от внешних фактров (предполагается, что в системе обеспечивается защита памяти процесса от других процессов, кроме случаев явного использования общей памяти). В коде тестов примеров макрос **assert()** используется и для проверки результатов системных функций, которые предполагаются всегда верно работающими в данном конкретном тесте. Это выглядит вполне разумно, ведь прогон тестов с отключенным макросом **assert()** явно является странной идеей, а продолжать выполнения теста в случае обнаружения неожиданной ошибки нет никакой нужды. С другой стороны, проверять результат всех системных функций на признак наличия ошибки совершенно необходимо даже в тестах.

### 3.4 Ошибки системных функций

Все системные функции, которые могут привести к ошибке, сообщают о ней своим результатом. Существуют как минимум следующие варианты результата, свидетельствующие об ошибке.

1) Возврат нулевого указателя, примеры: **strdup()**, **fopen()**;
2) Значение -1 в случае ошибки и значение 0 в случае успеха, примеры: **fork()**, **write()**.
3) Значение -1 в случае ошибки и иное значение в случае успеха, пример: **waitpid()**.
4) Значение, меньшее нуля — на практике это обычно именно -1, но стандарт допускает любое отрицательное. Примеры: **printf**.
5) Отличный от нуля код ошибки и значение 0 в случае успеха, причём значение **errno** не устанавливается в случае ошибки. Пример: **pthread_create()**.
6) Отличное от нуля значение в случае ошибки, пример: **dlclose()**.
7) Ноль в случае успеха, **EOF** в случае неудачи[1], пример: **fclose()**.
8) Любое значение типа **char** в случае успеха, **EOF** в случае неудачи[2], пример: **fgetc()**.

Как видно из списка, авторы различных системных функций POSIX были не слишком едины в воззрениях на значение результата в случае ошибки.

Сам код ошибки ни в одной системной функции не говорит о её конкретной причине. Для этого существует макрос **errno**. И функция **ferror**.

Кроме случая явной ошибки, инетересн случай, когда функция не выполнила цкликом ту работу, о которой её просили.

И сюда же про ЕИНТР.

### 3.5 Ошибки выделения памяти

Как известно, функции выделения памяти, такие **malloc()** или **strndup()**, могут вернуть значение **NULL** как признак невозможности выделить требуемую память. Хотя в современных системах с поддержкой виртуальной памяти, обработкой страничных исключений и наличием раздела на диске для подкачки в память такая ситуация маловероятна, разработчик ПО должен её отслеживать. Есть как минимум следующие сценарии, которые могут привести к ошибочке при вызове системных функций выделения памяти.

1) В разрабатываемом сервисе («демоне») наблюдается невыевленная утечка памяти, что приводит при его долгой работе исчерпанию памяти в сиетеме или исчерпанию виртуальной памяти процесса (последнее актуально для 32-х битных систем, где пользовательской части процесса выделяется обычно 2Гб виртуалной памяти).

[1] Обычно макрос **EOF** раскрывается в -1, но в стандарте это не оговорено.

[2] Обычно макрос **EOF** раскрывается в -1, но в стандарте это не оговорено.

2) Виртуальное адресное пространство фрагментруется в ходе его работы настолько, что уже нет непрерывного участка (актуально для 32-х битных систем).

3) Объем требуемой памяти рассчитан исходя из полученных процессом извне данных, которые могут ошибочными или даже полученными от злонамеренного источника.

4) Процесс выполняется в условиях исскуственно ограниченных ресурсов ().

5) Процесс выполняется в системе с ограниченными ресурсами, без подкачки памяти или даже на процессоре без MMU (характерный пример — коробочные сетевые устройства, недорогие ARM-платы для создания специализированных устройств).

Таким образом, существует довольно большое число случаев, когда написанный программистом код столкнется со случаем возврата функцией выделения памяти нулевого указателя. Вывод корректного сообщения об ошибке выделения памяти с указанием места её возникновения, очевидно, гораздо лучше, чем возникновение позже ошибки защиты памяти при последующей попытке разыменования нулевого указателя. Ошибку выделения не следует считать фатальной, столкнувшаяся с нею функция должна ...

К сожалению, различные POSIX-системы по разному реагируют при вызове функции **malloc()** с параметром, превыщающим возможности системы. Например, в ОС FreeBSD 9.0 вызывающий процесс будет просто уничтожен, а в ОС Debian 6.0 вызов вернет значение **NULL** и будет установленно значение **errno**. Таким образом, при написании модульных тестов бесполезно проверять, корректно ли работает функция при получении аргументов, преводящих к ошибке получения памяти, поскольку тестирующий процесс а некоторых системах будет убит, причем неигнорируемым сигналом **SIGKILL**. Для проверки поведения программы при ошибке выделения памяти поэтому нужно использовать свои варианты функций выделения памяти, которые будут возвращать нулевое значение. Для этого можно использовать, например, библиотеку dmalloc.

В завершении темы выделения памяти отметим, что в типичных системах функции выделения и освобождения памяти в основном реализованы в стандартой библиотеке. Их реализация запрашивает при необходимости у ядра виртуальную память целыми страницами, и потом разделяет выделенную процессу виртуальную память требуемыми программисту фрагментами. Термин «выделить виртуальную память» здесь означает, что ядро ОС отмечает в своих струтруха данных (в линуске это VMA) некоторую область виртуальной памяти в пользовательской части как «выделенную» и доступную для работы процесса, и при обращении процесса по этим адресам они должны быть корректно отображены ядром в физические адреса. Обращение к «невыделенной» виртуальной памяти считается ошибкой и приводит к уничтожению процесса (сингал **SIGSEGV**). Действующий стандарт POSIX не описывает стандартную функцию для выделения виртуальной памяти, которую могла бы использовать собственная реализация функции **malloc()**. Вы можете узнать, какой системный вызов используется конкретной библиотекой Си при помощи трассировщика системных вызовов.

### 3.6 Вывод сообщений об ошибках

Для вывода сообщений об ошибках исторически предназначется файловый дескриптор с номером 2 и связанная с ним файловая переменная с именем **stderr**. В наших примерах мы и воспользуемся ими для этой цели. При выводе сообщений в них полезно добавить разную слежубную информацию типа номера процесса, поэтому мы начнем с того, что напишем несколько макросов для этого. Поскольку хотелось бы сохранить возможность легко передавать затем в функцию **fprintf** переменное счисло аргументов, а также имя функции и номер строки в ней, где решено было вывести сообщение об ошибке, то нам нужно реализовать именно макросы, а не функцию.

Рассмотрим содержимое файла **errors.h** в проекте **common_macros**.

Листинг 3.3 — Неверное использование макроса **assert()**

```
#define warning(format, ...) do { \
    fprintf(stderr, "[WARNING] "format"\n", ##__VA_ARGS__); \
    fflush(stderr); \
} while (0)
```

.

На место специального символа **__VA_ARGS__** в результате работы препроцессора будут поставлены значения всех неименованные параметров макроса, разделенные запятыми. А что же такое **##**? Если ни одного такого параметра нет, то на место **__VA_ARGS__** не появится ни одного символа, и стоящая перед ним одинокая запятая приведет к ошибке компиляции. Таким образом, в нашем случае без использования конструкции **##** нужно будет иметь как минимум один переменный параметр при использовании макроса. Это не слишком удобно, и конструкция из двух символов решетки требует от препроцессора убрать идущую перед ними запятую, если после неё (что?). К сожалению, этот приём не вошел в стандарт языка Си, и является расширением компилятора (точнее, препроцессора) GCC. В настоящий момент компилятор CLang, как и некоторые другие, также поддерживают эту конструкцию.

Проверяйте через assert внутреннее состояние!

Проверять параметры как проверку утверждений — довольно спорное решение, поскольку при использовании проверки утверждений обычносложнее писать тесты. Кроме того, сложно сказать, насколько верным является условие в утверждении, возможно оно приводит к сложным срабатыванием (правда тут не ясно, чем обычная проверка лучше). Можно рекомендовать проверку параметров простым условным оператором и возвращать признак ошибки при неудаче проверки. Правда, этот способ лучше лишь тогда, когда вызывающая функция не забывает проверять результат функции на признак ошибки.

А вот промышленный код, возможно, стоит компилировать и с **NDEBUG**, чтобы отключить все утверждения (ведь на них-то не было тестов!).

Q. Назовите все возможные причины, по которым debug сделан макросом, а не функцией.

A. Удобно передавать аргументы в printf (иначе пришлось бы парсить командную строку и реализовывать printf самим, иначе никак), легче контролировать ошибки форматной строки (современные компиляторы обычно это проверяют).

### 3.7 Буфер для чтения данных

При чтении из файловых дескрипторов не гарантируется, что будет считан весь указанный буфер, даже если в ядре ОС накоплено достаточно данных. Кроме того, часто нам не известно, сколько байтов данных нам нужно считать: например, нам нужно обработать команды, и концом команды является байт переноса строки. В таком случае мы не можем заранее выделить буфер для всей считанной команды. С другой стороны, единократное чтение может считать несколько команд (причем последнюю — неполностью).

Даже в самой простой программе, использующий сокеты, между системными функциями чтения данных и функциями обработки полученных команд должен быть реализован промежуточный уровень, накапливающий считанные данные в буфер изменяемого размера, и выделяющий из него команды. (Опс. у нас буфер фиксированного размера. Надо исправить?) Мы заведем буфер для хранения полученных данных и будем вырезать из него команды по мере нахождения признака конца команды (символа **'\n'**). После вырезания команды мы будем смещать все оставшееся содержимое буфера в его начало (рисунок **??**). В функцию чтения данных будет передаваться указатель на первый свободный байт буфера. Отметим, что данные в буфере не завершаются нулевым символом.

Буфер реализован структурой **cuffer** и несколькими inline-функциями в файле **inc/buffer.h**.

Прокомментируем основные моменты. Функция инициализации (листинг 3.4 выделяет память для буфера и устанавливает значения текущей позиции для записи в него данных.

Листинг 3.4 — Инициализация буфера

```
struct buffer {
    char *buf;
    size_t size;
    size_t pos;
};

static inline
int init_buffer(struct buffer *buf, size_t size)
{
    buf->pos = 0;
    buf->buf = malloc(size);
    if (!buf->buf) {
        errnomsg("malloc()");
        buf->size = 0;
        return -1;
    }
    buf->size = size;
    return 0;
}
```

Функция **buffer_cut** вырезает указанное количество байт с начала буфера, выделяя для них память, остаток буфера сдвигается. Вырезанная последовательность дополняется нулевым байтом и является корректной строкой (конечно, если только среди записанных в буфер байтов не было нулевого байта).

Листинг 3.5 — Инициализация буфера

```
static inline
char *buffer_cut(struct buffer *buf, size_t len)
{
    if (buf->pos < len)
        return NULL;
    char *cut = malloc(len + 1);
    if (!cut) {
        errnomsg("malloc()");
        return NULL;
    }
    memcpy(cut, buf->buf, len);
    cut[len] = '\0';
    memmove(buf->buf, buf->buf + len, buf->pos - len);
    buf->pos -= len;
    return cut;
}
```

**buffer_shift**

Листинг 3.6 — Инициализация буфера

```
static inline
ssize_t buffer_shift(struct buffer *buf, size_t shift)
{
    if (buf->size < buf->pos + shift)
        return -1;
    return buf->pos += shift;
}
```

#### 3.7.0.1 Использование буфера изменяющегося размера

Реализуйте динамическое изменение размера буфера в файле . Для этого при его полном заполнение при отсутствии в нём признака конца команды размер буфера следует увеличить на четверть начального размера, если размер буфера еще меньше максимального. За максимальный размер можно взять начальный размер, умноженный на $2^k, k \in \rangle 2, 3, 4 \langle$. Отсутствие огранчиния роста размера буфера негативно скажется на работе системы в случае...

Добавьте в **tests/run.c** тест, проверяющий увеличение размера буфера и его прекращение при достижении максимального размера.

### 3.8 Работа с файловами

#### 3.8.1 Использование файловых дескрипторов

#### 3.8.2 Буфферизированный ввод-вывод

#### 3.8.3 Стандартные открытые дескрипторы

В POSIX-системах

Таблица 3.1 — Стандартные

**stdin**
**stdout**
**stderr**

## 4 Создание процессов

### 4.1 Программные библиотеки

#### 4.1.1 Архивы объектных файлов

**ar**

#### 4.1.2 Разделяемые библиотеки

#### 4.1.3 Проблема перемещаемых символов

Параметр **-fPIC**

### 4.2 Мультиплексирование чтения из файловых дескрипторов

#### 4.2.1 Особенности

Могут быть установлены одновременно события чтения и события завершения соединения. Можно предположить, что мы сначала можем проверить доступность данных и считать их, а потом поверить, нет ли события **POLLHUP**. Однако, не ясно, сколько данных при этом накопились в файловом дескрипторе. Разумеется, мы не можем полагаться, что нашего буфера хватит, чтобы считать их все за один вызов **read()**. Хуже того, стандарт и не гарантирует, что эти данные будут считаны за единственный вызов **read()** даже при достаточном размере буфера. Казалось бы, мы можем тогда обрабатывать событие **POLLHUP** только тогда, когда данных уже нет, а до тех пор — игнорировать его, но ведь никто не гарантирует, что проигнорированное событие **POLLHUP** будет в поле **revents** и при следующем вызове, а несчитанные данные ещё останутся.

По указанным соображениям мы воспользуемся наиболее безопасным путём — установим файловые дескрипторы в неблокирующий режим, и при событии **POLLIN** будем читать

из них в цикле, пока не получим значение **EAGAIN** в значении ошибки **errno**. После завершения цикла такого неблокирующего чтения мы можем проверить HUP и обработать его, закрыв файловый дескриптор.

Чем меньше внутри лока, тем лучше. лучше тупо в начале и конце одной функции, хотя формально это и не требуется.

## Заключение

## Список использованных источников

1. *Стивенс, У.Р.* UNIX. Профессиональное программирование / У.Р. Стивенс, С.А. Рago. — М: Символ-Плюс, 2007. — С. 1040.

## Приложение А Стиль оформления исходных текстов примеров

В этом приложении приведены основные положения стиля оформления исходный текстов программ-примеров на языке Си, используемый в пособии.

### Имена

Для имён файлов, функций, переменных, меток, структур, перечислений используются только строчные буквы, отдельные слова разделяются подчеркиваниями. Имена функции, когда это удобно, начинаются с глагола (примеры: **remove_connection**, **is_queue_empty**). За отсутствием в языке Си пространств имён имена ряда функций начинаются со слова, задающее «пространство имён», например: **buffer_shift** (файл **buffer.h**), **wrbuffer_shift** (файл **wrbuffer.h**).

Для имён макросов используются сплошные заглавные буквы в тех случаях, когда важно показать, что это макрос (например,**CHECK_ENITR**). Для ряда макросов с параметрами используются строчные буквы (например, **errnomsg**).

### Отступы и переносы строк

Отступ — четыре пробела. Открывающаяся фигурная скобка ставится на новой строке, если это начало функции, и в конце предыдущей строки во всех остальных случаях. Закрывающая фигурная скобка находится на отдельной строке, если только за ней не идёт слово **else**.

При определении функции её имя начинается с новой строки, это позволяет использовать для посика её определения команду типа TODO: **grep ^function_name -RI ***.

### Прочее

Определение типов (**typedef**) используется только для указателей на функции, имена таких типов заканчиваются на **_t**.