24.03.2025

Системные вызовы и ядра. Строковые алгоритмы. Vim – основы.

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА

Давайте познакомимся

Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

Адженда

Системные вызовы и ядра

35 минут

Строковые алгоритмы

30 минут

Vim – основы

25 минут

Адженда

Системные вызовы и ядра

35 минут

Строковые алгоритмы

30 минут

Vim – основы

25 минут

Системные вызовы

Исследование системных вызовов

Правило: содержимое двух внутренних колец, ядра и аппаратного обеспечения не должно быть доступно напрямую из пространства пользователя. Иными словами, никакой пользовательский процесс не должен иметь прямого доступа к оборудованию, а также к внутренним структурам данных и алгоритмам ядра. Обращение к ним должно происходить через системные вызовы.

Чтобы прояснить ситуацию, можно рассмотреть пример того, как данные проходят от пространства пользователя к пространству ядра и обратно. Когда вам нужно прочитать файл с жесткого диска, вы пишете программу для кольца пользовательских приложений. Она задействует функцию ввода/вывода из libc под названием fread (или ее аналог) и в итоге выполняется в виде процесса в пространстве пользователя. Когда программа вызывает функцию fread, срабатывает ее реализация внутри libc.

Пока все это происходит в пользовательском процессе. В конечном счете реализация fread инициирует системный вызов, принимая три аргумента: дескриптор уже открытого файла, адрес буфера, выделенного в памяти процесса (который находится в пространстве пользователя) и длину данного буфера.

Когда реализация libc инициирует системный вызов, поток выполнения пользовательского процесса переходит к ядру, которое получает аргументы из пользовательского пространства и размещает их в пространстве ядра. Затем ядро читает файл, обращаясь к своему модулю файловой системы.

Исследование системных вызовов

После завершения операции read в кольце ядра прочитанные данные копируются в буфер в пространстве пользователя, указанный во втором аргументе функции fread; дальше поток выполнения переходит от системного вызова к пользовательскому процессу. Пока системный вызов делает свою работу, пользовательский процесс обычно находится в ожидании. В таком случае системный вызов называют блокирующим.

Этот сценарий имеет несколько важных аспектов.

- Выполнением всей логики системных вызовов занимается одно ядро.
- Если системный вызов блокирующий, то вызывающая сторона должна дождаться завершения его работы. Неблокирующие системные вызовы очень быстро возвращаются, но пользовательскому процессу приходится выполнять дополнительные действия, чтобы проверить, доступны ли результаты.
- Аргументы вместе с входными и выходными данными копируются в пользовательское пространство и из него. Ввиду этого системные вызовы должны быть рассчитаны на прием крошечных значений и указателей в качестве входных аргументов.
- Ядро обладает полным привилегированным доступом ко всем ресурсам системы. Следовательно, должен существовать такой механизм, который проверяет, может ли пользовательский процесс выполнять тот или иной системный вызов.
- Похожее разделение существует между участками памяти, выделенными для пространств пользователя и ядра. Пользовательский процесс может обращаться только к памяти пространства пользователя. В ходе работы некоторых системных вызовов поток выполнения передается несколько раз.

Выполнение системного вызова напрямую, в обход стандартной библиотеки C

Пример, в котором системный вызов инициируется напрямую, в обход стандартной библиотеки C. Иными словами, программа выполняет системный вызов, не обращаясь к кольцу командной оболочки. Это считается плохим подходом, однако некоторые системные вызовы недоступны в libc и их пользовательское приложение может инициировать напрямую.

```
// Это нужно, чтобы использовать элементы не из состава POSIX
#define _GNU_SOURCE
#include <unistd.h>
// Это не является частью POSIX!
#include <sys/syscall.h>
int main(int argc, char **argv) {
    char message[20] = "Hello World!\n";
    // Инициирует системный вызов 'write', который записывает
    // некоторые байты в стандартный вывод.
    syscall(__NR_write, 1, message, 13);
    return 0;
}
```

```
$ gcc main.c -o a.out
$ ./a.out
Hello World!
```

Любая Unix-система предоставляет определенный метод для прямого выполнения системных вызовов. Например, в Linux для этого предусмотрена функция под названием syscall, размещенная в заголовочном файле <sys/syscall.h>.

В примере показана еще одна разновидность программы Hello World, в которой для передачи текста в стандартный вывод не используется libc. То есть не применяется функция printf, которая является частью кольца командной оболочки и стандарта POSIX. Вместо этого программа инициирует системный вызов напрямую, из-за чего данный код совместим только с Linux, но не с другими Unix-системами.

Выполнение системного вызова напрямую, в обход стандартной библиотеки C

```
$ strace ./a.out
execve("./a.out", ["./a.out"], 0x7ffc3cafcc60 /* 26 vars */) = 0
brk(NULL)                               = 0x55f2800a1000
arch_prctl(0x3001 /* ARCH_??? */, 0x7fff203c6960) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fd89ffbc000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
...
arch_prctl(ARCH_SET_FS, 0x7fd89fd87740) = 0
...
mprotect(0x7fd89ffa0000, 16384, PROT_READ) = 0
mprotect(0x55f27ee78000, 4096, PROT_READ) = 0
mprotect(0x7fd89fff6000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7fd89ffb3000, 34891)           = 0
write(1, "Hello World!\n", 13Hello World!)          = 13
exit_group(0)                           = ?
+++ exited with 0 +++
```

Здесь полужирным шрифтом выделено то место, где утилита strace записала системный вызов. Взгляните на возвращаемое значение (13). Оно означает, что системный вызов успешно записал 13 байт в заданный файл (в нашем случае это стандартный вывод).

Внутри функции syscall

```
#include <sysdep.h>
   /* Please consult the file sysdeps/unix/sysv/linux/x86-64/sysdep.h
for more information about the value -4095 used below.  */
   /* Usage: long syscall (syscall_number, arg1, arg2, arg3, arg4,
arg5, arg6)   We need to do some arg shifting, the syscall_number will
be in rax.  */
        .text
   ENTRY (syscall)
        movq %rdi, %rax      /* Syscall number -> rax.  */
        movq %rsi, %rdi      /* shift arg1 - arg5.  */
        movq %rdx, %rsi
        movq %rcx, %rdx
        movq %r8, %r10
        movq %r9, %r8
        movq 8(%rsp),%r9 /* arg6 is on the stack.  */
        syscall          /* Do the system call.  */
        cmpq $-4095, %rax    /* Check %rax for error.  */
        jae SYSCALL_ERROR_LABEL  /* Jump to error handler if error. */
        ret          /* Return to caller.  */

   PSEUDO_END (syscall)
```

Этот метод выполнения системного вызова более сложен, но сами инструкции выглядят лаконично и просто.

glibc не позволяет использовать определенные возможности ядер, которые поддерживают системные вызовы с более чем шестью аргументами и добавление такой поддержки потребует изменения данной библиотеки. К счастью, шести аргументов достаточно в большинстве случаев.

В листинге выше вслед за инструкцией movq ассемблерный код вызывает процедуру syscall. Он просто генерирует прерывание, которое активирует определенный участок ядра, предназначенный для его обработки.

Что касается функции syscall, то ее объявление написано на C, а определение — на ассемблере.

Добавление системного вызова в Linux

Опасность! Повторяйте только на виртуальных машинах (например, VirtualBox!)

Oracle VirtualBox Менеджер

Файл Машина Справка

Инструменты

Создать Добавить Настроить Сбросить Запустить

nuclei_change
Выключена

Общие
Имя: nuclei_change
ОС: Ubuntu (64-bit)

Система
Оперативная память: 4096 МБ
Процессоры: 4
Порядок загрузки: Жёсткий диск, Оптический диск, Гибкий диск
Ускорение: Nested Paging, Паравиртуализация KVM

Дисплей
Видеопамять: 128 МБ
Графический контроллер: VMSVGA
Сервер удалённого дисплея: Выключен
Запись: Выключена

Носители
Контроллер: IDE
Вторичное устройство IDE 0: [Оптический привод] Пусто
Контроллер: SATA
SATA порт 0: nuclei_change.vdi (Обычный, 29,96 ГБ)

Аудио
Аудиодрайвер: По умолчанию
Аудиоконтроллер: ICH AC97

Сеть
Адаптер 1: Intel PRO/1000 MT Desktop (NAT)

USB
USB-контроллер: OHCI, EHCI
Фильтры устройств: 0 (0 активно)

Общие папки
Отсутствуют

Включение ВМ ...
0%
nuclei_change

Разработка ядра

Разработка ядер имеет ряд отличий от того, как разрабатываются обычные программы на C. Шесть ключевых отличий в процессе разработки для ядра и для пользовательского пространства:

- Процесс ядра находится во главе системы и существует в единственном экземпляре. Это просто означает, что если ваш код приведет к сбою в ядре, то вам, скорее всего, придется перезагрузить компьютер и позволить ядру заново выполнить инициализацию.
- В кольце ядра нет никакой стандартной библиотеки C наподобие glibc! Иными словами, стандарты SUS и POSIX здесь не действуют. Например, занимаясь разработкой ядра в Linux, вы можете выполнять запись в буфер сообщений ядра с помощью функции printk. Но в FreeBSD для этого предусмотрено семейство функций printf, которые отличаются от своих аналогов из libc; они находятся в заголовочном файле <sys/system.h>.
- В ядре тоже можно читать и изменять файлы, но без использования функций из libc. У каждой Unix-системы есть собственный способ доступа к файлам из кольца ядра.
- В кольце ядра вы имеете полный доступ к физической памяти и многим другим возможностям.
- В ядре нет механизма системных вызовов. Системные вызовы — основной метод взаимодействия с кольцом ядра из пользовательского пространства. Поэтому в самом ядре в нем нет необходимости.
- Процесс ядра создается путем копирования образа ядра в физическую память. Этим занимается загрузчик. Вы не можете добавить новые системные вызовы, не прибегнув к созданию совершенно нового образа и инициализации его в ходе перезагрузки системы. Кое-какие ядра поддерживают модули, которые можно динамически добавлять и удалять, но с системными вызовами этого делать нельзя.

Все это показывает, что процесс разработки ядра отличается от обычной разработки на языке C. Тестирование написанной логики затруднено, и некачественный код может вывести из строя всю систему.

Написание демонстрационного системного вызова для Linux

Здесь мы напишем новый системный вызов для Linux. В Интернете есть много материала на эту тему, но мы возьмем за основу статью Adding a Hello World System Call to Linux Kerne (через VPN)!

Прежде всего мы должны загрузить самый свежий исходный код ядра Linux. Мы клонируем его из репозитория на GitHub и укажем нужный нам выпуск. **Linux** — это ядро. То есть оно может быть установлено только в кольце ядра Unix-подобной операционной системы. А вот дистрибутив Linux — другое дело. Он содержит определенные версии ядра Linux, библиотеки GNU libc и командной оболочки Bash (или GNU Shell). Дистрибутивы Linux обычно поставляются с набором пользовательских приложений во внешних кольцах, поэтому их можно считать полноценными операционными системами. Обратите внимание: дистрибутив, «дистр» и разновидность Linux — одно и то же.

```
$ sudo apt-get update
$ sudo apt-get install -y build-essential autoconf libncurses5-dev
libssl-dev bison flex libelf-dev git
...
...
```

apt — основной диспетчер пакетов в дистрибутивах Linux, основанных на Debian, а sudo — утилита, которая используется для запуска команд от имени администратора. Они доступны почти во всех Unix-подобных операционных системах.

Написание демонстрационного системного вызова для Linux

```
$ git clone https://github.com/torvalds/linux
$ cd linux
$ git checkout v5.3
```

Некоторые из этих каталогов могут показаться знакомыми: fs, mm, net, arch и т. д. Они могут существенно отличаться в зависимости от ядра; смысл в том, что большинство ядер имеют одну и ту же внутреннюю структуру.

Итак, получив исходники ядра, мы можем приступить к созданию нашего нового системного вызова. Но сначала нужно выбрать для него уникальный числовой идентификатор; в данном случае я назову его hello_world и назначу ему номер 999.

```
$ ls
total 760K
drwxrwxr-x 33 mikhail mikhail 4.0K Jan 28 2018 arch
drwxrwxr-x 3 mikhail mikhail 4.0K Oct 16 22:11 block
drwxrwxr-x 2 mikhail mikhail 4.0K Oct 16 22:11 certs
...
drwxrwxr-x 75 mikhail mikhail 4.0K Oct 16 22:11 fs
drwxrwxr-x 27 mikhail mikhail 4.0K Jan 28 2018 include
...
drwxrwxr-x 17 mikhail mikhail 4.0K Oct 16 22:11 kernel
drwxrwxr-x 13 mikhail mikhail 12K Oct 16 22:11 lib
-rw-rw-r-- 1 mikhail mikhail 429K Oct 16 22:11 MAINTAINERS
-rw-rw-r-- 1 mikhail mikhail 61K Oct 16 22:11 Makefile
drwxrwxr-x 3 mikhail mikhail 4.0K Oct 16 22:11 mm
drwxrwxr-x 69 mikhail mikhail 4.0K Jan 28 2018 net
-rw-rw-r-- 1 mikhail mikhail 722 Jan 28 2018 README
...
drwxrwxr-x 4 mikhail mikhail 4.0K Jan 28 2018 virt
drwxrwxr-x 5 mikhail mikhail 4.0K Oct 16 22:11 zfs
$
```

Написание демонстрационного системного вызова для Linux

Объявление нового системного вызова Hello World (include/linux/syscalls.h)

```
/*
 * syscalls.h - Linux syscall interfaces (non-arch
 *
 * Copyright (c) 2004 Randy Dunlap
 * Copyright (c) 2004 Open Source Development Labs
 * This file is released under the GPLv2.
 * See the file COPYING for more details.
 */
#ifndef _LINUX_SYSCALLS_H
#define _LINUX_SYSCALLS_H
struct epoll_event;
struct iattr;
struct inode;
...
asmlinkage long sys_statx(int dfd, const char __user *path, unsigned flags,
                          unsigned mask, struct statx __user *buffer);
asmlinkage long sys_hello_world(const char __user *str,
                                const size_t str_len,
                                char __user *buf,
                                size_t buf_len);

#endif
```

Согласно описанию это заголовочный файл с интерфейсами syscall, не зависящими от архитектуры. Это значит, Linux предоставляет один и тот же набор системных вызовов на всех архитектурах.
В конце файла мы объявили функцию нашего системного вызова, которая принимает четыре аргумента. Обратите внимание: входные аргументы обозначены как const, а выходные — нет. Кроме того, идентификатор __user означает, что указатели ссылаются на адреса в пользовательском пространстве. В нашем системном вызове 0 означает успех, а любое другое число — провал.

Написание демонстрационного системного вызова для Linux

```
#include <linux/kernel.h>   // Для printk
#include <linux/string.h>   // Для strcpy, strcat, strlen
#include <linux/slab.h>     // Для kmalloc, kfree
#include <linux/uaccess.h>  // Для copy_from_user, copy_to_user
#include <linux/syscalls.h> // Для SYSCALL_DEFINE4
SYSCALL_DEFINE4(hello_world, // Определение системного вызова
                const char __user *, str,    // Вводимое имя
                const unsigned int, str_len, // Длина вводимого имени
                char __user *, buf,          // Выходной буфер
                unsigned int, buf_len) {     // Длина выходного буфера
    // Переменная в стеке ядра должна хранить содержимое
    // входного буфера
    char name[64];
    // Переменная в стеке ядра должна хранить итоговое
    // выходное сообщение.
    char message[96];
    ...
    strcpy(message, "Hello "); // Формируем итоговое сообщение
    strcat(message, name);
    strcat(message, "!");
    // Проверяем, помещается ли итоговое сообщение в выходной двоичный буфер
    ...
    // Записываем отправленное сообщение в журнал ядра
    printk("Message: %s\n", message);
    return 0;
}
```

Теперь нам нужно определить наш системный вызов. Для этого необходимо сначала создать папку с именем hello_world в корневом каталоге.
Создание каталога hello_world
$ mkdir hello_world
$ cd hello_world $
Затем внутри hello_world нужно создать файл с именем sys_hello_world.c. Он должен иметь следующее содержимое

Написание демонстрационного системного вызова для Linux

Следующим шагом будет обновление еще одной таблицы. В архитектурах x86 и x64 предусмотрена лишь одна таблица для системных вызовов; мы должны добавить в нее наш новый системный вызов, чтобы сделать его доступным. Только после этого системный вызов можно будет использовать на компьютерах с архитектурами x86 и x64. Итак, мы должны добавить в таблицу системный вызов hello_word и имя его функции, sys_hello_world.

Откройте для этого файл arch/x86/entry/syscalls/syscall_64.tbl и добавьте в его конец следующую строчку.

Добавление нового системного вызова Hello World в таблицу системных вызовов

```
999   64   hello_world     __x64_sys_hello_world
```

После внесения этого изменения файл должен выглядеть так.

Системный вызов Hello World был добавлен в таблицу системных вызовов

```
$ cat arch/x86/entry/syscalls/syscall_64.tbl
...
...
546 x32 preadv2 __x32_compat_sys_preadv64v2
547 x32 pwritev2 __x32_compat_sys_pwritev64v2
999 64 hello_world __x64_sys_hello_world
```

Написание демонстрационного системного вызова для Linux

Для компиляции всех исходных файлов и создания итогового образа ядра Linux используется система сборки Make. Вы должны создать файл Makefile в каталоге hello_world. Этот файл должен содержать одну строчку следующего вида.

Файл Makefile для системного вызова Hello World

```
obj-y := sys_hello_world.o
```

Затем вам нужно добавить каталог hello_world в главный файл Makefile, который находится в корневом каталоге. Перейдите в корневой каталог ядра, откройте Makefile и найдите следующую строчку.

Строчка, которую необходимо изменить в корневом файле Makefile

```
core-y += kernel/certs/mm/fs/ipc/security/crypto/block/
```

Добавьте в данный список hello_world/. Это всего лишь перечень каталогов, которые должны собираться вместе с ядром.

Нам нужно добавить каталог системного вызова Hello World, чтобы он был включен в процесс сборки и стал частью итогового образа ядра. После редактирования эта строчка должна выглядеть следующим образом.

Строчка после редактирования

```
core-y += kernel/certs/mm/fs/hello_world/ipc/security/crypto/block/
```

Сборка ядра

Чтобы собрать ядро, нам нужно сначала вернуться в его корневой каталог и предоставить конфигурацию со списком возможностей и модулей, которые будут скомпилированы в процессе сборки.

Команда пытается сгенерировать нужную нам конфигурацию на основе включенной в текущее ядро Linux. Она берет имеющуюся конфигурацию и при наличии в нашем ядре, которое мы пытаемся собрать, новых значений просит вас их подтвердить. Для подтверждения достаточно нажать клавишу Enter.

Создание новой конфигурации на основе текущего активного ядра

```
$ make localmodconfig
...
...
#
# configuration written to .config
#
$
```

Сборка ядра

Теперь можно начать процесс сборки. Поскольку ядро Linux содержит много исходных файлов, сборка может занять несколько часов. Поэтому компиляцию нужно выполнять параллельно.

Если вы используете виртуальную машину, то, пожалуйста, сделайте ее процессор многоядерным. Это существенно ускорит процесс сборки.

```
$ make -j4
SYSHDR arch/x86/include/generated/asm/unistd_32_ia32.h
SYSTBL arch/x86/include/generated/asm/syscalls_32.h
HOSTCC scripts/basic/bin2c
SYSHDR arch/x86/include/generated/asm/unistd_64_x32.h
...
UPD include/generated/compile.h
CC init/main.o
CC hello_world/sys_hello_world.o
CC arch/x86/crypto/crc32c-intel_glue.o
...
LD [M] sound/pci/ac97/snd-ac97-codec.ko
LD [M] sound/pci/snd-intel8x0.ko
LD [M] sound/soundcore.ko
$
```

Сборка ядра

Не забудьте установить все необходимые пакеты, представленные в первой части этого раздела, иначе получите ошибки компиляции.

Как видите, начался процесс сборки, состоящий из четырех заданий, которые пытаются скомпилировать файлы C в параллель.

Создание и установка нового образа ядра

```
$ sudo make modules_install install
INSTALL arch/x86/crypto/aes-x86_64.ko
INSTALL arch/x86/crypto/aesni-intel.ko
INSTALL arch/x86/crypto/crc32-pclmul.ko
INSTALL arch/x86/crypto/crct10dif-pclmul.ko
...
run-parts: executing /et/knel/postinst.d/initam-tools 5.3.0+ /
boot/vmlinuz-5.3.0+
update-iniras: Generating /boot/initrd.img-5.3.0+
run-parts: executing /etc/keneostinst.d/unattende-urades 5.3.0+ /
boot/vmlinuz-5.3.0+
...
Found initrd image: /boot/initrd.img-4.15.0-36-generic
Found linux image: /boot/vmlinuz-4.15.0-29-generic
Found initrd image: /boot/initrd.img-4.15.0-29-generic done.
$
```

Сборка ядра

Новый образ ядра версии 5.3.0 был создан и установлен. Теперь мы можем перезагрузить компьютер. Но прежде не забудьте проверить версию текущего ядра, если не знаете ее.

Проверка версии текущего ядра

```
$ uname -r
4.15.0-36-generic
$
```

Теперь перезагрузим систему.

Перезагрузка системы

```
$ sudo reboot
```

В ходе загрузки системы будет выбрано и задействовано новое ядро. Стоит отметить, что загрузчик не подхватит старые ядра; следовательно, если у вас было ядро версии 5.3 и выше, то вам придется загрузить собранный вами образ вручную. Когда операционная система завершит загрузку, у вас должно быть новое ядро.

Проверка версии ядра после перезагрузки

```
$ uname -r
5.3.0+
$
```

Если все прошло как следует, то у вас должно быть загружено новое ядро.

Сборка ядра

Теперь мы можем вернуться к написанию программы на C, которая использует наш только что добавленный системный вызов Hello World.

```c
// Это нужно для использования элементов, не входящих в состав POSIX
#define _GNU_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>// Это не является частью POSIX!
int main(int argc, char **argv) {
    char str[20] = "Kam";
    char message[64] = "";
    // Инициируем системный вызов hello world
    int ret_val = syscall(999, str, 4, message, 64);
    if (ret_val < 0) {
        printf("[ERR] Ret val: %d\n", ret_val);
        return 1;
    }
    printf("Message: %s\n", message);
    return 0;
}
```

```
$ gcc main.c -o main.out
$ ./main.out
Message: Hello Kam!
$
```

Как видите, мы инициировали системный вызов под номер 999. Мы подали на вход строку Kam и ожидаем получить в качестве приветствия Hello Kam!. Программа ожидает получения результата и выводит буфер с сообщением, записанный системным вызовом в пространстве ядра.

Сборка ядра

Использование команды dmesg для просмотра журнальных записей, сгенерированных системным вызовом Hello World

```
$ dmesg
...
[ 112.273783] System call fired!
[ 112.273786] Message: Hello Kam!
```

Мониторинг системных вызовов, инициированных примером

```
$ strace ./main.out
...
mprotect(0x55726020000, 4096, PROT_READ) = 0
mprotect(0x7f8dd6d2d000, 4096, PROT_READ) = 0
munmap(0x7f8dd6d26000, 27048) = 0
syscall_0x3e7(0x7fffe7d2af30, 0x4, 0x7fffe7d2af50, 0x40,
0x7f8dd6b01d80, 0x7fffe7d2b088) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
brk(NULL) = 0x5572674f2000
brk(0x557267513000)
...
exit_group(0) = ?
+++ exited with 0 +++
```

Сборка ядра

Передача длинного сообщения (больше 64 байт) системному вызову Hello World

```
int main(int argc, char** argv) {
    char name[84] = "A very very long message! It is really hard to
    produce a big string!";
    char message[64] = "";
    ...
    return 0;
}
```

Компиляция и запуск отредактированного примера

```
$ gcc main.c -o main.out
$ ./main.out
[ERR] Ret val: -1
```

Мониторинг системных вызовов

```
$ strace ./main.out
...
munmap(0x7f1a900a5000, 27048) = 0
syscall_0x3e7(0x7ffdf74e10f0, 0x54, 0x7ffdf74e1110, 0x40,
0x7f1a8fe80d80, 0x7ffdf74e1248) = -1 (errno 1)
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
brk(NULL) = 0x5646802e2000
...
exit_group(1) = ?
+++ exited with 1 +++
```

Ядра Unix

В этом разделе речь пойдет об архитектурах, которые были разработаны в ядрах Unix за последние 30 лет. Но прежде, чем обсуждать разные виды ядер, которых, к слову, не так уж и много, следует отметить, что процесс проектирования ядра никак не стандартизован.

Каждое ядро по-своему уникально. Но все они имеют общую черту: предоставляют доступ к своим возможностям через интерфейс системных вызовов. Однако каждое ядро имеет собственное видение того, как нужно обрабатывать эти вызовы.

Такое разнообразие и споры вокруг него стали одной из самых горячих тем в области компьютерных архитектур в 1990-х годах. В этих обсуждениях принимали участие большие группы людей. Самым знаменитым примером считаются дебаты между Таненбаумом и Торвальдсом.

Два основных направления в проектировании ядер Unix: монолит и микроядро. Существуют и другие архитектуры, такие как гибридные ядра, наноядра и экзоядра, и каждая из них имеет определенное назначение.

Монолитные ядра и микроядра

Монолитное ядро представляет собой один процесс с единым адресным пространством, содержащий более мелкие компоненты. Микроядра используют противоположный подход. ***Микроядро*** — минимальный процесс, из которого в пользовательское пространство вынесены такие компоненты, как файловая система, драйверы устройств и управление процессами; благодаря этому процесс ядра получается более компактным.

Обе архитектуры имеют свои преимущества и недостатки, благодаря чему стали темой одного из самых знаменитых обсуждений в истории операционных систем. Все началось еще в 1992 году, сразу после выпуска первой версии Linux. Начало дискуссии положил Эндрю С. Таненбаум публикацией в сети Usenet.

Публикация породила непримиримое противостояние между Таненбаумом, создателем Linux, Линусом Торвальдсом, и многими другими энтузиастами, которые позже стали одними из первых разработчиков Linux. Споры велись о природе монолитных ядер и микроядер. Они касались таких тем, как архитектура ядра и влияние на нее аппаратной платформы.

Монолитные ядра и микроядра

Ниже перечислены различия между монолитными ядрами и микроядрами.

- Монолитное ядро представляет собой единый процесс, содержащий все возможности, предоставляемые этим ядром. У микроядер все иначе: каждую возможность они предоставляют в виде отдельного процесса.
- Процесс монолитного ядра находится в пространстве ядра, тогда как в микроядре серверные процессы обычно вынесены в пространство пользователя. Серверными называют процессы, которые предоставляют доступ к компонентам ядра, таким как диспетчер памяти, файловая система и т. д.
- Монолитные ядра, как правило, быстрее. Это объясняется тем, что все их функции выполняются внутри одного процесса. Микроядрам же нужно организовывать обмен сообщениями между пространствами пользователя и ядра, что приводит к большему количеству системных вызовов и переключений контекста.
- В монолитной архитектуре все драйверы устройств загружаются прямо в ядро. С микроядрами все иначе, поскольку все драйверы устройств и многие другие компоненты выполняются в пространстве пользователя.
- Для взлома монолитного ядра и, следовательно, всей системы достаточно внедрить в него небольшой фрагмент вредоносного кода.
- В монолитной архитектуре малейшее изменение исходного кода требует перекомпиляции всего ядра и создания его нового образа.

Один из самых известных примеров микроядра — MINIX. Этот проект, написанный Эндрю С. Таненбаумом, изначально создавался в образовательных целях.

Linux

Вы уже познакомились с ядром Linux, мы разрабатывали новый системный вызов. Здесь же мы сделаем акцент на монолитности Linux и на том факте, что в процессе ядра находятся все его возможности.

Однако должен существовать способ добавить в ядро новую функциональность, не прибегая к его перекомпиляции. Этого нельзя достичь за счет новых системных вызовов, поскольку, как вы уже видели, для их добавления нужно отредактировать много важных файлов, а это означает компиляцию ядра заново.

Но есть другой подход. Он предусматривает написание и динамическое подключение к ядру модулей. Именно о модулях пойдет речь в следующем подразделе, а затем мы попробуем написать собственный модуль для Linux.

Модули ядра

Монолитные ядра обычно предусматривают механизм, который позволяет динамически подключать к активному ядру новые компоненты. Эти подключаемые компоненты называются модулями ядра.

В отличие от серверных процессов в микроядрах, которые фактически являются отдельными процессами, взаимодействующими между собой по IPC, модули представляют собой объектные файлы ядра, скомпилированные и готовые к загрузке в его процесс. Эти файлы можно собрать статически и сделать их частью образа ядра; но вы также можете загружать их динамически, когда ядро уже работает.

Обратите внимание: по своей сути объектные файлы ядра аналогичны объектным файлам, которые создаются в процессе разработки на языке C.

В целом общение с модулями ядра в Linux и некоторых похожих операционных системах происходит тремя способами.

- Файлы устройств в каталоге /dev. Модули ядра в основном разрабатываются для использования в качестве драйверов устройств. Вот почему самый распространенный способ взаимодействия с ними — использование файлов внутри каталога /dev.
- Содержимое procfs. Файлы в каталоге /proc можно использовать для чтения метаинформации об отдельных модулях ядра. Эти файлы также позволяют отправлять модулям метаинформацию или управляющие команды.
- Содержимое sysfs. Это еще одна файловая система в Linux, которая позволяет управлять пользовательскими процессами и другими компонентами, относящимися к ядру (включая модули ядра), с помощью скриптов или вручную. Ее можно считать новой версией procfs.

Создание нового модуля ядра для Linux

Здесь мы напишем новый модуль ядра для Linux под названием Hello World. Он будет создавать запись в procfs. Затем мы используем эту запись, чтобы прочитать строку с приветствием.

Вы научитесь писать, компилировать и загружать модули в ядро, выгружать их оттуда и читать данные из записи в procfs. ***Основное назначение этого примера*** — дать вам возможность набраться опыта и стать более самостоятельным разработчиком.

Модули ядра компилируются в объектные файлы, которые можно загружать непосредственно в активное ядро. Для этого не нужно перезагружать систему. Главное, чтобы ваш модуль не натворил беды в ядре и не вывел его из строя. То же самое можно сказать о выгрузке модуля.

Первым делом нужно создать каталог, в котором будут находиться все файлы, относящиеся к модулю ядра.

```
$ mkdir example
$ cd example
$
```

Модуль ядра Hello World (example/hwkm.c) (24_3)

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
// Структура, ссылающаяся на файл proc
// Функция обратного вызова для чтения файла
// Обратный вызов для инициализации модуля
// Обратный вызов для выхода из модуля
// Определение обратных вызовов для работы с модулем
```

Создание нового модуля ядра для Linux

Обратите внимание: на данном этапе код нашего модуля имеет неограниченный доступ к почти любым компонентам ядра и из него в пользовательское пространство может утечь какая угодно информация. Это серьезная дыра в безопасности, вследствие чего вам следует поближе познакомиться с рекомендациями по написанию безопасных модулей ядра.

Чтобы скомпилировать приведенный выше код, нам нужен подходящий компилятор, с помощью которого мы потенциально можем выполнить компоновку с нужными нам библиотеками. Чтобы упростить себе жизнь, создадим файл с именем Makefile, который вызовет все необходимые инструменты для сборки нашего модуля ядра. ***Содержимое файла Makefile для модуля ядра Hello World***

```
obj-m += hwkm.o
all:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Создание нового модуля ядра для Linux

Сборка модуля ядра Hello World

```
$ make
make -C /lib/modules/54.318.0+/build M=/home/m/example
modules
make[1]: Entering directory '/home/m/linux'
CC [M] /home/m/example/hwkm.o
Building modules, stage 2.
MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /home/m/example/hwkm.o
see include/linux/module.h for more information
CC /home/m/hwkm.mod.o
LD [M] /home/m/hwkm.ko
make[1]: Leaving directory '/home/m/linux'
```

Как видите, в результате компиляции кода получается объектный файл, который затем компонуется с другими библиотеками в файл .ko. Среди сгенерированных результатов должен находиться файл с именем hwkm.ko.

Обратите внимание на расширение .ko. Оно означает, что выходной файл является объектным файлом ядра. Это нечто вроде разделяемой библиотеки, которую можно динамически загружать в ядро, где она будет выполняться.

Создание нового модуля ядра для Linux

Список существующих файлов после сборки модуля ядра Hello World

```
$ ls -l
total 556
-rw-rw-r-- 1 m m 154 Oct 19 00:36 Makefile
-rw-rw-r-- 1 m m 0 Oct 19 08:15 Module.symvers
-rw-rw-r-- 1 m m 1104 Oct 19 08:05 hwkm.c
-rw-rw-r-- 1 m m 272280 Oct 19 08:15 hwkm.ko
-rw-rw-r-- 1 m m 596 Oct 19 08:15 hwkm.mod.c
-rw-rw-r-- 1 m m 104488 Oct 19 08:15 hwkm.mod.o
-rw-rw-r-- 1 m m 169272 Oct 19 08:15 hwkm.o
-rw-rw-r-- 1 m m 54 Oct 19 08:15 modules.order
$
```

Мы использовали инструменты сборки модулей из ядра Linux версии 5.3.0. Вы можете получить ошибку, если попытаетесь скомпилировать этот пример с помощью ядра, версия которого ниже 3.10.

Загрузка и установка модуля ядра Hello World

```
$ sudo insmod hwkm.ko
$
```

Создание нового модуля ядра для Linux

Проверка журнальных сообщений ядра после установки модуля

```
$ dmesg
...
...
[ 7411.519575] Hello World module is loaded.
$
```

Итак, модуль загружен, и уже должен быть создан файл /proc/hwkm. Мы можем прочитать его с помощью команды cat. ***Чтение файла /proc/hwkm с использованием команды cat***

```
$ cat /proc/hwkm
Hello World From Kernel Module!
$ cat /proc/hwkm
Hello World From Kernel Module!
```

Выгрузка модуля ядра Hello World

```
$ sudo rmmod hwkm
```

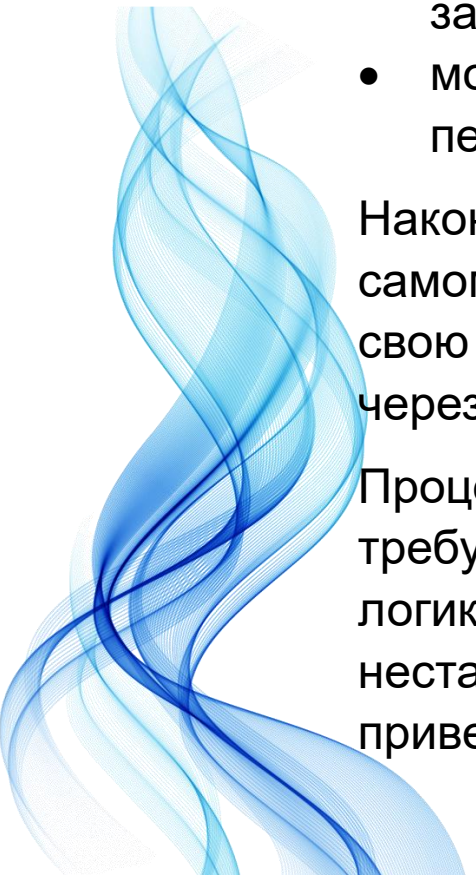
Проверка журнальных сообщений ядра после выгрузки модуля

```
$ dmesg
...
...
[ 7411.519575] Hello World module is loaded.
[ 7648.950639] Goodbye World!
```

Создание нового модуля ядра для Linux

Особенности модулей ядра, с которыми мы успели познакомиться:

- модули ядра можно загружать и выгружать, не прибегая к перезагрузке компьютера;
- после загрузки модули становятся частью ядра и получают доступ ко всем его компонентам и структурам. Это можно считать уязвимостью, но ядро Linux имеет средства защиты от загрузки нежелательных модулей;
- модули ядра можно компилировать отдельно. А вот системные вызовы требуют перекомпиляции всего ядра, на что может легко уйти несколько часов.



Наконец, модули ядра могут пригодиться при разработке кода, который необходимо выполнять в самом ядре и инициировать с помощью системных вызовов. Сначала вы можете реализовать свою логику в виде модуля, а затем, протестировав ее должным образом, сделать доступной через настоящий системный вызов.

Процесс разработки системных вызовов с нуля может быть довольно трудоемким, поскольку требует регулярной перезагрузки компьютера. Но если сначала написать и протестировать логику в виде модуля, то это упростит разработку ядра. ***Стоит отметить:*** если ваш код нестабильный, то неважно, где он находится — в модуле или в системном вызове; он может привести к сбою в ядре, что потребует перезагрузки системы.

Адженда

Системные вызовы и ядра

35 минут

Строковые алгоритмы

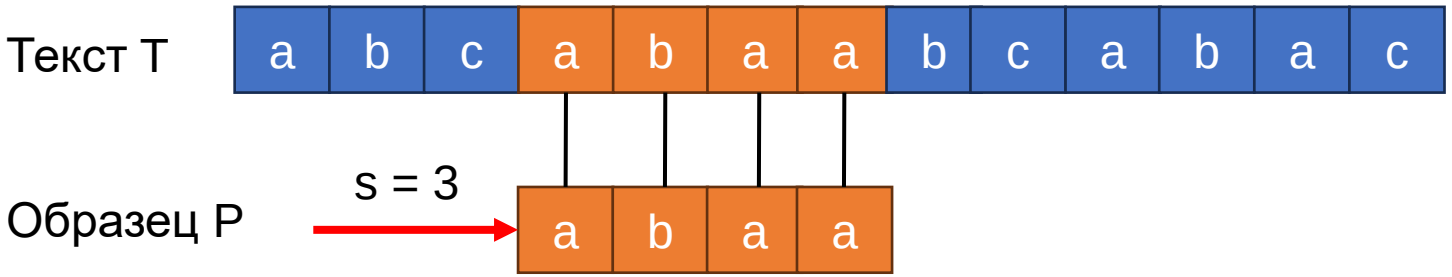
30 минут

Vim – основы

25 минут

Введение

В программах, предназначенных для редактирования текста, часто необходимо найти все фрагменты текста, которые совпадают с заданным образцом. Обычно **текст** — это редактируемый документ, а **образец** — это искомое слово, введенное пользователем. Эффективные алгоритмы решения этой задачи (часто именуемой сопоставлением строк (string matching)) могут сокращать время реакции текстовых редакторов на действия пользователя. Среди множества приложений алгоритмы поиска подстрок используются, например, для поиска заданных образцов в молекулах ДНК. Поисковые системы в Интернете также используют их при поиске веб-страниц, отвечающих запросам.



Алгоритм	Время предварительной обработки	Время сравнения
"В лоб"	0	O((n - m + 1)m)
Рабина Карпа	Θ(m)	O((n - m + 1)m)
Конечный автомат	O(m|Σ|)	Θ(n)
Кнута-Морриса-Пратта	Θ(m)	Θ(n)

Постановку задачи поиска подстрок (string-matching problem):
Предполагается, что текст задан в виде массива T[1…n] длиной n, а образец (шаблон) - в виде массива P[1…m] длиной m ≤ n. Далее, предполагается, что элементы массивов P и T - символы из конечного алфавита Σ. Например, алфавит может иметь вид Σ = {0,1} или Σ = {a,b,…, z}. Символьные массивы P и T часто называют строками (strings) символов.

Обозначения и терминология

Обозначим через Σ^* множество всех строк конечной длины, образованных с помощью символов алфавита Σ. ***Пустая строка*** (empty string), которая обозначается ε и имеет нулевую длину, также принадлежит множеству Σ^*. Длина строки x обозначается $|x|$. ***Конкатенация*** (concatenation) строк x и y, которая обозначается xy, имеет длину $|x| + |y|$ и состоит из символов строки x, после которых следуют символы строки y.

Строка w является ***префиксом*** (prefix) строки x (обозначается как $w \sqsubset x$), если $x = wy$ для некоторой строки $y \in \Sigma^*$. Заметим, что если $w \sqsubset x$, то $|w| \leq |x|$.

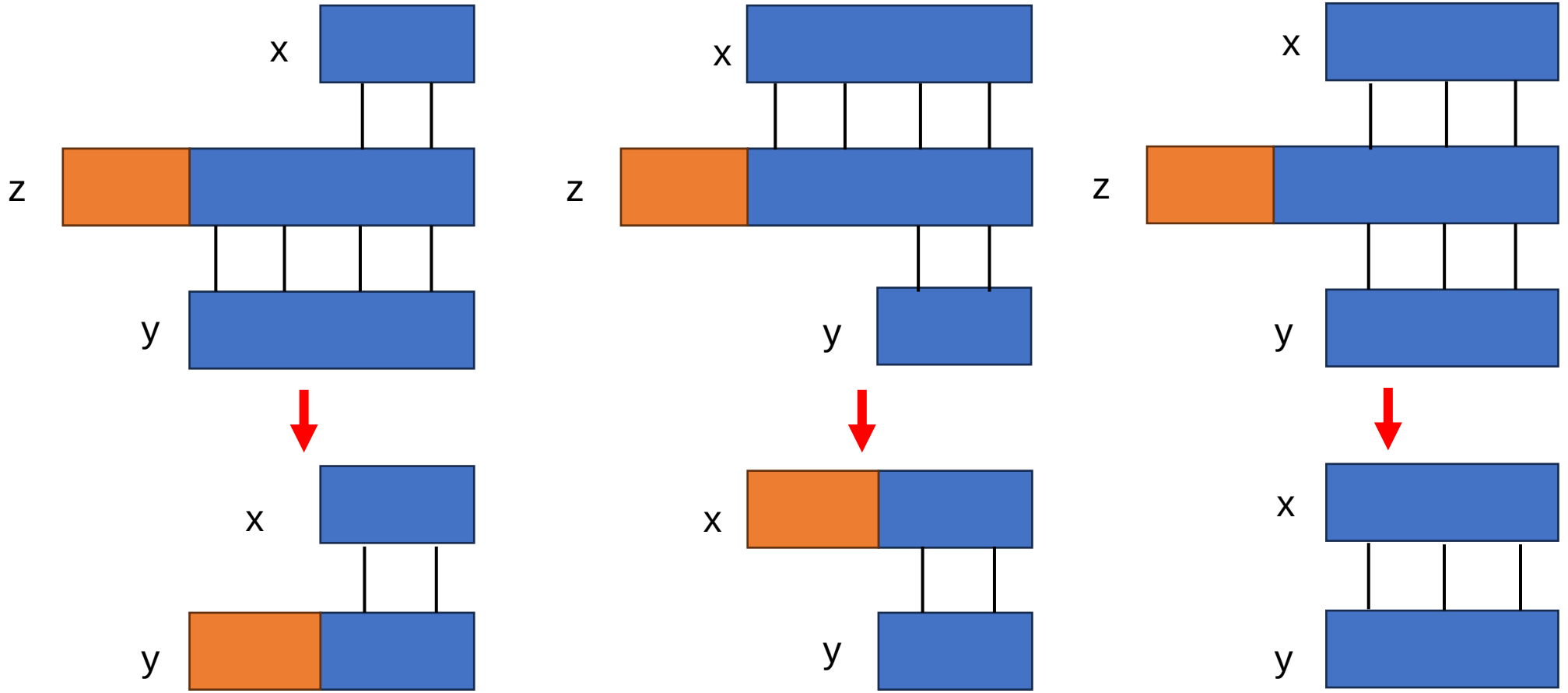
Строку w называют ***суффиксом*** (suffix) строки x (обозначается как $w \sqsupset x$), если существует такая строка $y \in \Sigma^*$, что $x = yw$. Как и в случае префикса, из w x следует неравенство $|w| \leq |x|$.

Например, $ab \sqsubset abcca$ и $cca \sqsupset abcca$.

Пустая строка ε является одновременно и суффиксом, и префиксом любой строки. Для произвольных строк x и y и для любого символа a соотношение $x \sqsupset y$ выполняется тогда и только тогда, когда $xa \sqsupset ya$. Кроме того, заметим, что отношения $\sqsubset$ и $\sqsupset$ являются транзитивными.

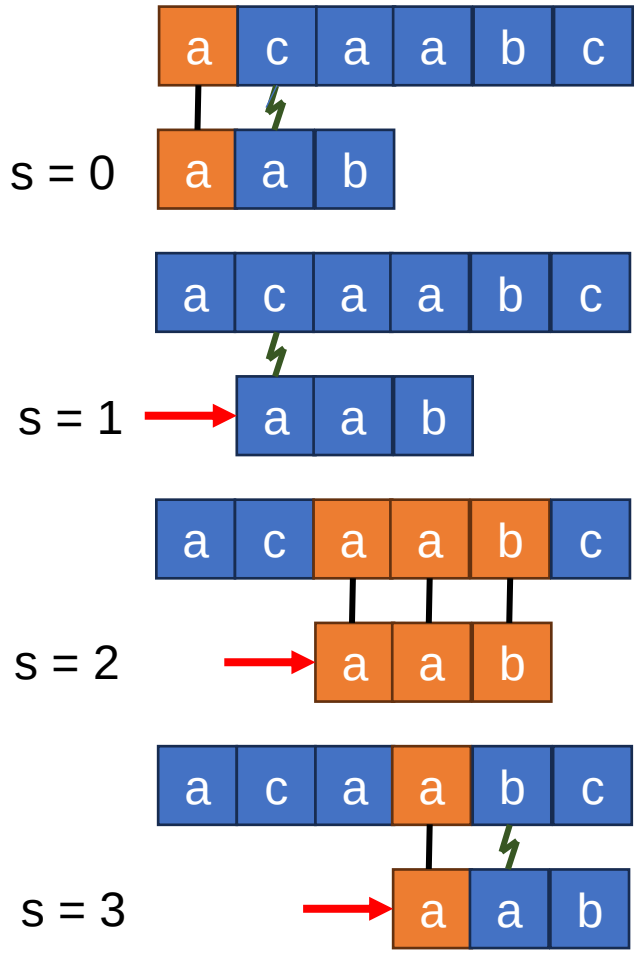
Обозначения и терминология

Лемма (Лемма о перекрывающихся суффиксах)
Предположим, что x, y и z являются строками, такими, что x ⊐ z и y ⊐ z. Если |x| ≤ |y|, то x ⊐ y. Если |x| ≥ |y|, то y ⊐ x. Если |x| = |y|, то x = y.



Простейший алгоритм поиска подстрок

```
void NaiveStringMatcher(const char *T, const char *P) {
    int n = strlen(T);
    int m = strlen(P);
    for (int s = 0; s <= n - m; s++) {
        int match = 1;
        for (int i = 0; i < m; i++) {
            if (P[i] != T[s + i]) {
                match = 0;
                break;
            }
        }
        if (match)
            printf("Образец найден со сдвигом %d\n", s);
    }
}
```



Время работы процедуры NaiveStringMatcher равно O((n - m + 1)m), и это точная оценка в наихудшем случае.

Алгоритм Рабина-Карпа

Рабин (Rabin) и Карп (Karp) предложили алгоритм поиска подстрок, показывающий на практике хорошую производительность, а также допускающий обобщения на другие родственные задачи, такие как задача о сопоставлении двумерного образца. В алгоритме Рабина-Карпа время $\Theta(m)$ затрачивается на предварительную обработку, а время его работы в наихудшем случае оказывается равным $\Theta((n - m + 1)m)$. Однако с учетом определенных предположений удается показать, что среднее время работы этого алгоритма существенно лучше.

Для простоты ***предположим,*** что $\Sigma = \{0, 1, 2, \ldots, 9\}$, т.е. что каждый символ представляет собой десятичную цифру. (В общем случае каждый символ — это цифра в системе счисления с основанием d, где $d = |\Sigma|$.)

Для заданного образца **P[1…m]** обозначим через **p** соответствующее ему десятичное значение. Аналогично для заданного текста **T[1…n]** обозначим через $\mathbf{t_s}$ десятичное значение подстрок $T[s + 1 \ldots s + m]$ длиной m при $s = 0, 1, \ldots, n - m$.

Если бы значение p можно было вычислить за время $\Theta(m)$, а все значения t_s - за суммарное время $\Theta(n - m + 1)$, то значения всех допустимых сдвигов можно было бы определить за время $\Theta(m) + \Theta(n - m + 1) = \Theta(n)$ путем сравнения значения p с каждым из значений t_s.

Алгоритм Рабина-Карпа

С помощью правила Горнера величину p можно вычислить за время Θ(m):
$p = P[m] + 10 (P[m - 1] + 10(P[m - 2] + … + 10(P[2] + 10P[1])…))$.
Аналогично t_0 можно вычислить из T[1…m] за то же время Θ(m). Чтобы вычислить остальные значения t_1, t_2, …, t_{n-m} за время Θ(n - m), заметим, что величину t_{s+1} можно вычислить из величины t_s за константное время, так как
$t_{s+1} = 10(t_s - 10^{m-1}T[s + 1]) + T[s + m + 1]$.
Вычитание $10^{m-1}T[s + 1]$ удаляет старшую цифру из t_s, умножение полученного результата на 10 сдвигает число влево на один десятичный разряд, а добавление T[s + m + 1] вносит в него соответствующую младшую цифру.

Пример

m = 5 и $t_s = 31415$, то необходимо удалить старшую цифру T[s + 1] = 3 и добавить новую младшую цифру (предположим, это T[s + 5 + 1] = 2), чтобы получить
$t_{s+1} = 10(31415 - 10000 • 3) + 2 = 14152$.
Таким образом, число p можно вычислить за время Θ(m), величины t_0, t_1, …,t_{n-m} - за время Θ(n - m + 1), а все вхождения образца P[1…m] в текст T[1…n] можно найти, затратив на фазу предварительной обработки время Θ(m), а на фазу сравнения - время Θ(n - m + 1).

Алгоритм Рабина-Карпа

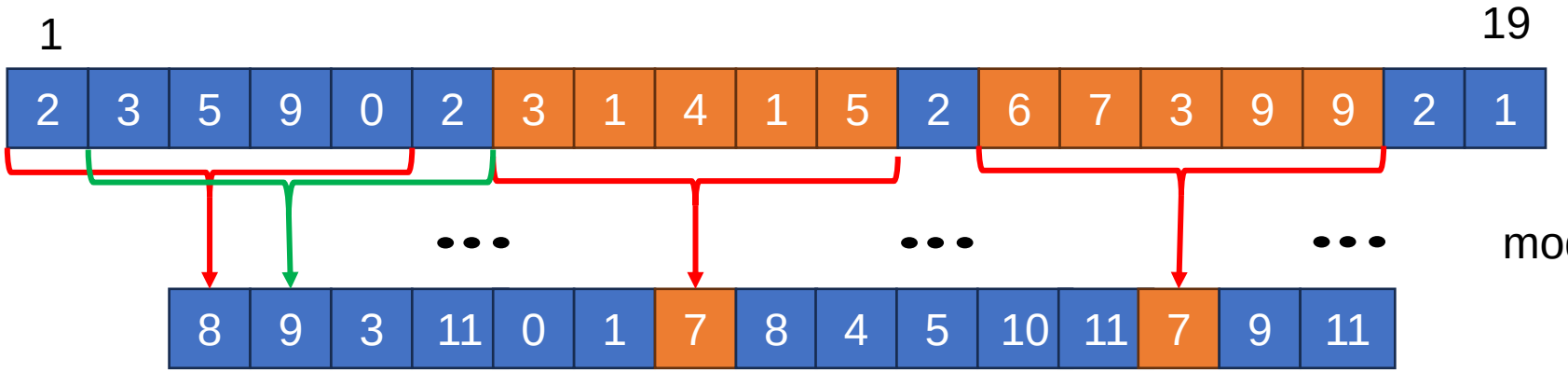
Единственная сложность, возникающая в этой процедуре, может быть связана с тем, что значения p и t_s могут оказаться слишком большими и с ними будет неудобно работать. Если образец P содержит m символов, то предположение о том, что каждая арифметическая операция с числом p (в которое входит m цифр) занимает "фиксированное время", не отвечает действительности. К счастью, эта проблема имеет простое решение: вычислять значения p и t по модулю некоторого числа q. Значение p по модулю q можно вычислить за время Θ(m), а все значения t, по модулю q - за время Θ(n - m + 1). Если выбрать в качестве значения q такое простое число, что 10q помещается в компьютерное слово, то все необходимые вычисления можно выполнить с использованием арифметики одинарной точности. В общем случае, если имеется d-символьный алфавит {0, 1, …, d - 1}, значение q выбирается таким образом, чтобы величина dq помещалась в компьютерное слово, и рекуррентное соотношение исправляется так, чтобы оно работало по модулю q, т.е. записываем его в виде

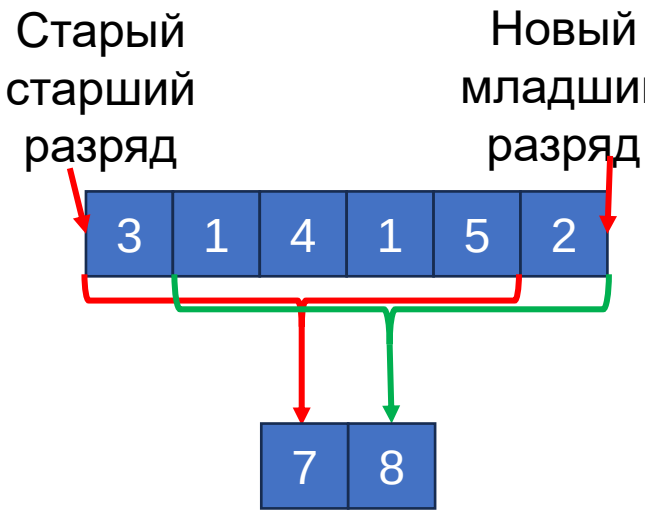
$$t_{s+1} = (d(t_s - T[s + 1]h) + T[s + m + 1]) \bmod q ,$$

где $h = d^{m-1} \pmod q$ - значение цифры "1" в старшем разряде окна размером m цифр.

Алгоритм Рабина-Карпа

P = 31415





$$14152 \equiv (31415 - 10000 \cdot 3) \cdot 10 + 2 \pmod{13}$$
$$\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13}$$
$$\equiv 8 \pmod{13}$$

Алгоритм Рабина-Карпа 1/2

```
#include <stdio.h>
#include <string.h>
// Функция для вычисления (a * b) mod m
long long modMultiply(long long a, long long b, long long mod) {
    return (a * b) % mod;
}
// Функция для вычисления (a + b) mod m
long long modAdd(long long a, long long b, long long mod) {
    return (a + b) % mod;
}
```

```
void RabinKarpMatcher(const char *T, const char *P, int d, int q)
{
    int n = strlen(T); // Длина текста
    int m = strlen(P); // Длина образца
    long long h = 1;   // Вычисляем h = d^(m-1) mod q
    for (int i = 1; i < m; i++)
        h = modMultiply(h, d, q);
    long long p = 0; // Вычисляем хэш образца p
    for (int i = 0; i < m; i++)
        p = modAdd(modMultiply(d, p, q), P[i], q);
```

Алгоритм Рабина-Карпа 2/2

```
    long long t = 0; // Вычисляем начальный хэш текста t0
    for (int i = 0; i < m; i++)
        t = modAdd(modMultiply(d, t, q), T[i], q);
    for (int s = 0; s <= n - m; s++) {    // Поиск совпадений
        if (p == t) { // Если хэши совпадают
            int match = 1;
            // Проверяем точное совпадение
            for (int i = 0; i < m; i++) {
                if (P[i] != T[s + i]) {
                    match = 0;
                    break;
                }
            }
            if (match)
                printf("Образец найден со сдвигом %d\n", s);
        }
        if (s < n - m) {// Вычисляем новый хэш для следующей подстроки
            t = modAdd(modMultiply(d, (t - T[s] * h + q) % q, q), T[s + m], q);
            t = (t + q) % q; // Убеждаемся, что t неотрицательное
        }
    }
}
```

Конечные автоматы

Конечный автомат M представляет собой кортеж из пяти значений $(Q, q_0, A, \Sigma, \delta)$, где

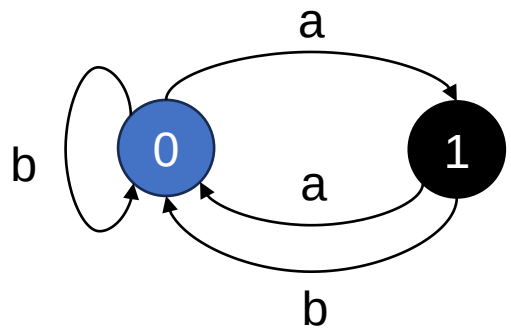
- Q - конечное множество состояний;
- $q_0 \in Q$ - ***начальное состояние***;
- $A \subseteq Q$ - множество различных ***допускающих состояний***;
- Σ - конечный входной алфавит;
- δ - функция, отображающая $Q \times \Sigma$ в Q и называемая ***функцией переходов автомата*** M.

Вначале конечный автомат находится в состоянии q_0 и считывает символы входной строки по одному. Если автомат находится в состоянии q и считывает входной символ a, он совершает переход из состояния q в состояние $\delta(q, a)$. Если текущее состояние q является членом A, то машина M ***принимает***, или ***допускает*** (accepted), строку, считанную к этому моменту. Не принятые входные данные являются отвергнутыми (rejected). С конечным автоматом M связана функция φ, которая называется ***функцией конечного состояния*** (final state function) и отображает множество Σ^* на множество Q, так, что значение $\varphi(\omega)$ представляет собой состояние, в котором оказывается автомат M после сканирования строки ω. Таким образом, M принимает строку w тогда и только тогда, когда $\varphi(\omega) \in A$. Функция φ определяется следующим рекуррентным соотношением с использованием функции переходов:

$\varphi(\varepsilon) = q_0$

$\varphi(\omega a) = \delta(\varphi(\omega), a)$ для $\omega \in \Sigma^*$, $a \in \Sigma$.

Состояние	Вход a	Вход b
0	1	0
1	0	0



Автоматы поиска подстрок

Для каждого образца P строится свой автомат поиска подстрок; его необходимо сконструировать по образцу на этапе предварительной обработки. С этого момента предполагается, что образец P — это заданная фиксированная строка: для краткости мы будем опускать в обозначениях зависимость от P.

Чтобы создать автомат поиска подстрок, соответствующий заданному образцу P[1…m], сначала определим вспомогательную функцию σ, которая называется ***суффиксной функцией*** (suffix function), соответствующей образцу P. Функция σ является отображением множества Σ* на множество {0,1, …,m}, таким, что величина σ(x) равна длине максимального префикса P, который является суффиксом строки x:

$$\sigma(x) = \max\{k: P_k \sqsupset x\} \qquad (2.3)$$

Автоматы поиска подстрок

Суффиксная функция σ вполне определена, поскольку пустая строка $P_0 = \varepsilon$ является суффиксом любой строки. В качестве примера рассмотрим образец P = ab; тогда σ(ε) = 0, σ(cacca) = 1 и σ(ccab) = 2. Для образца P длиной m равенство σ(x) = m выполняется тогда и только тогда, когда P ⊐ x. Из определения суффиксной функции следует, что если x ⊐ y, то σ(x) ≤ σ(y).

Определим автомат поиска подстрок, соответствующий образцу P[1…m], следующим образом.

- Множество состояний Q имеет вид {0, 1, …,m}. Начальным состоянием q_0 является состояние 0, а состояние m является единственным принимающим состоянием.
- Функция переходов δ определена для любого состояния q и символа, а уравнением

$$\delta(q, a) = \sigma(P_q a). \qquad (2.4)$$

Мы определяем $\delta(q, a) = \sigma(P_q a)$, потому что хотим отследить самый длинный префикс образца P, который до сих пор соответствовал текстовой строке T. Чтобы подстрока T - скажем, подстрока, заканчивающаяся в T[i] – соответствовала некоторому префиксу P_j образца P, этот префикс P_j должен быть суффиксом T_i. Предположим, что $q = \varphi(T_i)$, так что после чтения T_i автомат находится в состоянии q. Разработаем функцию переходов таким образом, чтобы этот номер состояния, q, указывал длину наибольшего префикса P, который соответствует суффиксу T_i, т.е. чтобы в состоянии q выполнялись соотношения $P_q \sqsupset T_i$ и $q = \sigma(T_i)$. (Когда q = m, все m символов P соответствуют суффиксу T_i, так что мы находим искомую подстроку.) Таким образом, поскольку и $\varphi(T_i)$, и $\sigma(T_i)$ равны q, мы увидим (ниже, в теореме 2.4), что автомат поддерживает следующий инвариант:

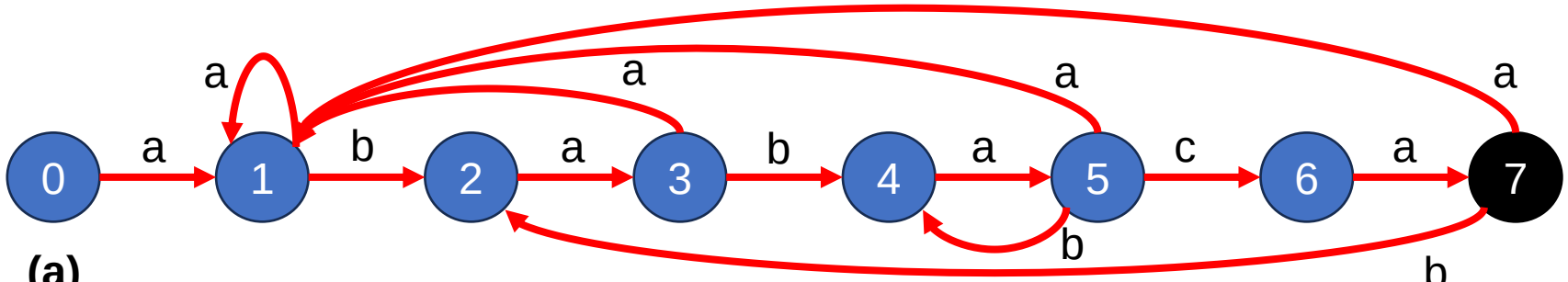
$$\varphi(T_i) = \sigma(T_i). \qquad (2.5)$$

Автоматы поиска подстрок

Если автомат находится в состоянии q и считывает очередной символ T[i+1] = a, то необходимо, чтобы переход вел в состояние, соответствующее наибольшему префиксу P, который одновременно является суффиксом $T_i a$, а этим состоянием является $\sigma(T_i a)$. Поскольку представляет собой наибольший префикс P, являющийся суффиксом T_i, наибольший префикс P, который является суффиксом $T_i a$, не только $\sigma(T_i a)$, но и $\sigma(P_q a)$. (Лемма 2.3 доказывает, что $\sigma(T_i a) = \sigma(P_q a)$.) Таким образом, необходимо, чтобы, когда автомат находится в состоянии q, функция переходов для символа a переводила автомат в состояние $\sigma(P_q a)$.

Следует рассмотреть два случая. В первом случае a = P[q + 1], так что символ a продолжает соответствие образцу; при этом, поскольку δ(q,a) = q + 1, переход продолжает выполняться вдоль "хребта" автомата. Во втором случае a ≠ P[q+1], так что a не соответствует образцу. Здесь нужно найти меньший префикс P, который одновременно является суффиксом T_i. Поскольку на этапе предварительной обработки при создании автомата образец сопоставляется с самим собой, функция переходов быстро находит наибольший среди таких меньших префиксов P.

Автоматы поиска подстрок



(а)

(б)

Состояние	Вход a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(в)

i	-	1	2	3	4	5	6	7	8	9	10	11
T[i]	-	a	b	a	b	a	b	a	c	a	b	a
Состояние $\varphi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(а) Диаграмма состояний автомата поиска подстрок, который воспринимает все строки, оканчивающиеся строкой ababaca. Состояние 0 - начальное, а состояние 7 (оно выделено черным цветом) - единственное допускающее.

(б) Соответствующая функция переходов δ, а также строка образца P = ababaca.

(в) Обработка автоматом текста T = abababacaba. Под каждым символом текста T[i] указано состояние φ(Ti), в котором находится автомат после обработки префикса Ti.

Автоматы поиска подстрок

```
// Основная функция Finite Automaton Matcher
void FiniteAutomatonMatcher(char T[], char P[], int m) {
    int n = strlen(T);  // Длина текста
    // Текущее состояние автомата
    for (int q = 0, i = 0; i < n; i++) {
        q = transitionFunction(q, T[i], P, m);
        if (q == m) {
            printf("Образец найден со сдвигом %d\n", i - m + 1);
        }
    }
}
```

Чтобы пояснить работу автомата, предназначенного для поиска подстрок, приведем простую, но эффективную программу для моделирования поведения такого автомата (представленного функцией переходов δ) при поиске образца P длиной m во входном тексте T[1…n].

Из простой циклической структуры процедуры Finite-Automaton-Matcher можно легко увидеть, что время поиска совпадений с его помощью в тексте длиной n равно Θ(n). Однако сюда не входит время предварительной обработки, которое необходимо для вычисления функции переходов δ. К этой задаче мы обратимся позже, после доказательства корректности работы процедуры Finite-Automaton-Matcher.

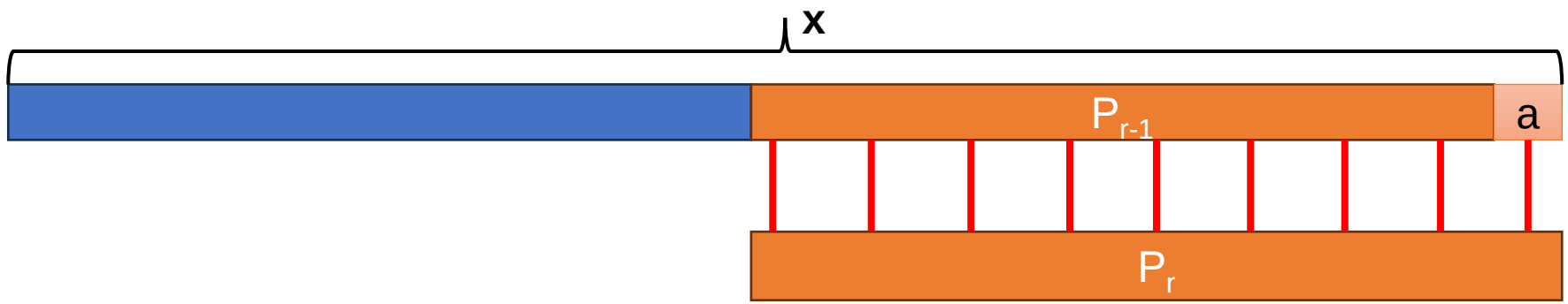
Рассмотрим, как автомат обрабатывает входной текст T[1…n]. Докажем, что после сканирования символа T[i] автомат окажется в состоянии $\sigma(T_i)$. Поскольку соотношение $\sigma(T_i) = m$ справедливо тогда и только тогда, когда $P \sqsupset T_i$, машина окажется в принимающем состоянии m тогда и только тогда, когда она только что считает образец P. Чтобы доказать этот результат, воспользуемся двумя приведенными ниже леммами, в которых идет речь о суффиксной функции σ.

Автоматы поиска подстрок

Лемма 2.2 (Неравенство суффиксной функции)

Для любой строки x и символа a выполняется неравенство $\sigma(xa) \le \sigma(x) + 1$.

Доказательство. Введем обозначение $r = \sigma(xa)$ (рисунок). Если $r = 0$, то неравенство $\sigma(xa) = r \le \sigma(x) + 1$ тривиальным образом следует из того, что величина $\sigma(x)$ неотрицательна. Теперь предположим, что $r > 0$. Тогда $P_r \sqsupset xa$ согласно определению функции σ. Если в конце строк P_r и xa отбросить по символу a, то получим $P_{r-1} \sqsupset x$. Следовательно, справедливо неравенство $r - 1 \le \sigma(x)$, так как $\sigma(x)$ - наибольшее значение k, при котором выполняется соотношение $P_k \sqsupset x$, и, таким образом, $\sigma(xa) = r \le \sigma(x) + 1$.



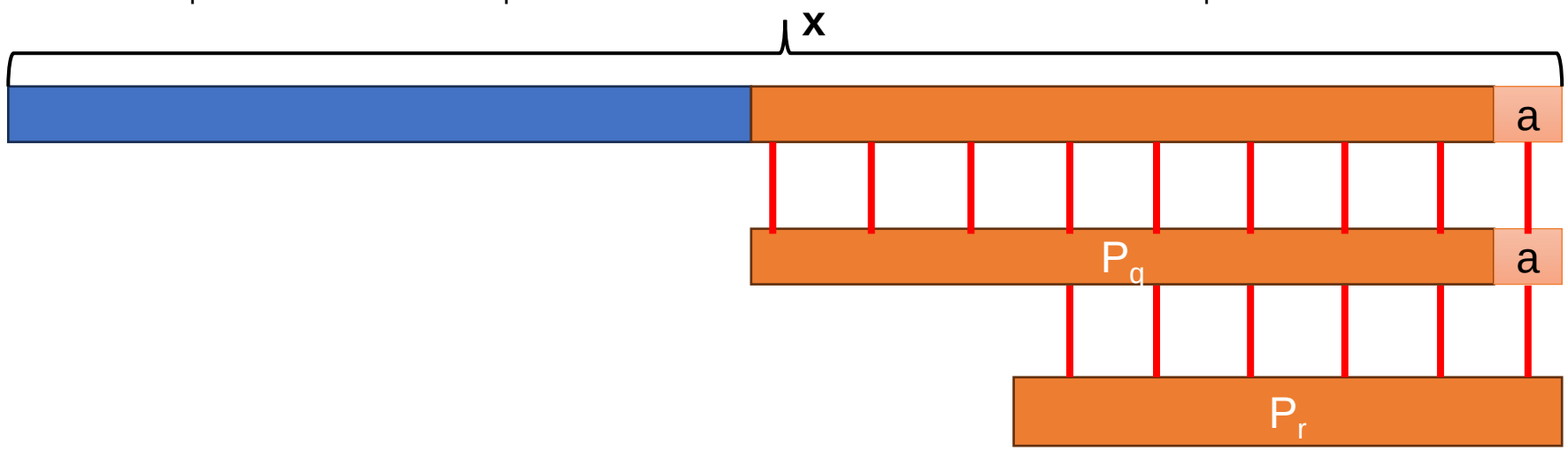
На рисунке показано, что $r \le \sigma(x) + 1$, где $r = \sigma(xa)$.

Автоматы поиска подстрок

Лемма 2.3 (Лемма о рекурсии суффиксной функции)

Если для произвольной строки x и символа a выполняется q = σ(x), то σ(xa) = σ(P_qa).

Доказательство. $P_q \sqsupset x$ из определения σ. Как показано на рисунке, выполняется также $P_q a \sqsupset xa$. Если обозначить r = σ(xa), то $P_r \sqsupset xa$ и, согласно лемме 2.2, r ≤ q+ 1. Таким образом, имеем $|P_r| = r \le q+1 = |P_q a|$. Поскольку $P_q a \sqsupset xa$, $P_r \sqsupset xa$ и $|P_r| \le |P_q a|$, из леммы 2.1 следует, что $P_r \sqsupset P_q a$. Следовательно, r ≤ σ(P_qa), т.е. σ(xa) ≤ σ(P_qa). Но, кроме того, σ(xa) ≥ σ(P_qa), поскольку $P_q a \sqsupset xa$. Таким образом, σ(xa) = σ(P_qa).



На рисунке показано, что r = σ(P_qa), где q = σ(x) и r = σ(xa).

Автоматы поиска подстрок

Теперь мы готовы доказать основную теорему, характеризующую поведение автомата поиска подстрок при обработке входного текста.

Теорема 2.4

Если φ - функция конечного состояния автомата поиска подстрок для заданного образца P, а T[1…n] - входной текст автомата, то $\varphi(T_i) = \sigma(T_i)$

Доказательство. Доказательство выполняется по индукции по i. Для i = 0 теорема тривиально истинна, поскольку $T_0 = \varepsilon$. Таким образом, $\varphi(T_0) = 0 = \sigma(T_0)$.

Теперь предположим, что $\varphi(T_i) = \sigma(T_i)$ и докажем, что $\varphi(T_{i+1}) = \sigma(T_{i+1})$. Обозначим $\varphi(T_i)$ как q, T[i + 1] - как a. Тогда

$\varphi(T_{i+1}) = \varphi(T_i a)$ (согласно определению T_{i+1} и a)

$= \delta(\varphi(T_i), a)$ (согласно определению φ)

$= \delta(q,a)$ (согласно определению q)

$= \sigma(P4a)$ (согласно определению δ (2.4))

$= \sigma(T_i a)$ (согласно лемме 2.3 и гипотезе индукции)

$= \sigma(T_{i+1})$ (согласно определению T_{i+1}).

Автоматы поиска подстрок

Теперь мы готовы доказать основную теорему, характеризующую поведение автомата поиска подстрок при обработке входного текста.

Теорема 2.4

Если φ - функция конечного состояния автомата поиска подстрок для заданного образца P, а T[1…n] - входной текст автомата, то $\varphi(T_i) = \sigma(T_i)$

Доказательство. Доказательство выполняется по индукции по i. Для i = 0 теорема тривиально истинна, поскольку $T_0 = \varepsilon$. Таким образом, $\varphi(T_0) = 0 = \sigma(T_0)$.

Теперь предположим, что $\varphi(T_i) = \sigma(T_i)$ и докажем, что $\varphi(T_{i+1}) = \sigma(T_{i+1})$. Обозначим $\varphi(T_i)$ как q, T[i + 1] - как a. Тогда

$\varphi(T_{i+1}) = \varphi(T_i a)$ (согласно определению T_{i+1} и a)

$= \delta(\varphi(T_i), a)$ (согласно определению φ)

$= \delta(q,a)$ (согласно определению q)

$= \sigma(P4a)$ (согласно определению δ (2.4))

$= \sigma(T_i a)$ (согласно лемме 2.3 и гипотезе индукции)

$= \sigma(T_{i+1})$ (согласно определению T_{i+1}).

Вычисление функции переходов

```
// Вычисление функции переходов
int **ComputeTransitionFunction(char *P, char *Sigma) {
    int m = strlen(P);              // Длина образца
    int sigmaSize = strlen(Sigma); // Размер алфавита
    int q, i, k;
    // Выделение памяти под таблицу переходов δ[q][a]
    int **delta = (int **)malloc((m + 1) * sizeof(int *));
    for (q = 0; q <= m; q++)
        delta[q] = (int *)malloc(sigmaSize * sizeof(int));
    // Заполнение таблицы переходов
    for (q = 0; q <= m; q++) {
        for (i = 0; i < sigmaSize; i++) {
            char a = Sigma[i];
            k = (m + 1 < q + 2) ? m + 1 : q + 2; // min(m + 1, q + 2)
            do {
                k--;
            } while (!isSuffix(P, k, q, a));
            delta[q][i] = k;
        }
    }
    return delta;
}
```

В этой процедуре функция $\delta(q, a)$ вычисляется непосредственно, исходя из ее определения (2.4). Время работы процедуры Compute-Transition-Function равно $O(m^3|\Sigma|)$, поскольку внешние циклы дают вклад, соответствующий умножению на $m|\Sigma|$, внутренний цикл repeat может повториться не более $m + 1$ раз, а для выполнения проверки $P_k \supset P_q a$ может потребоваться сравнить вплоть до m символов.

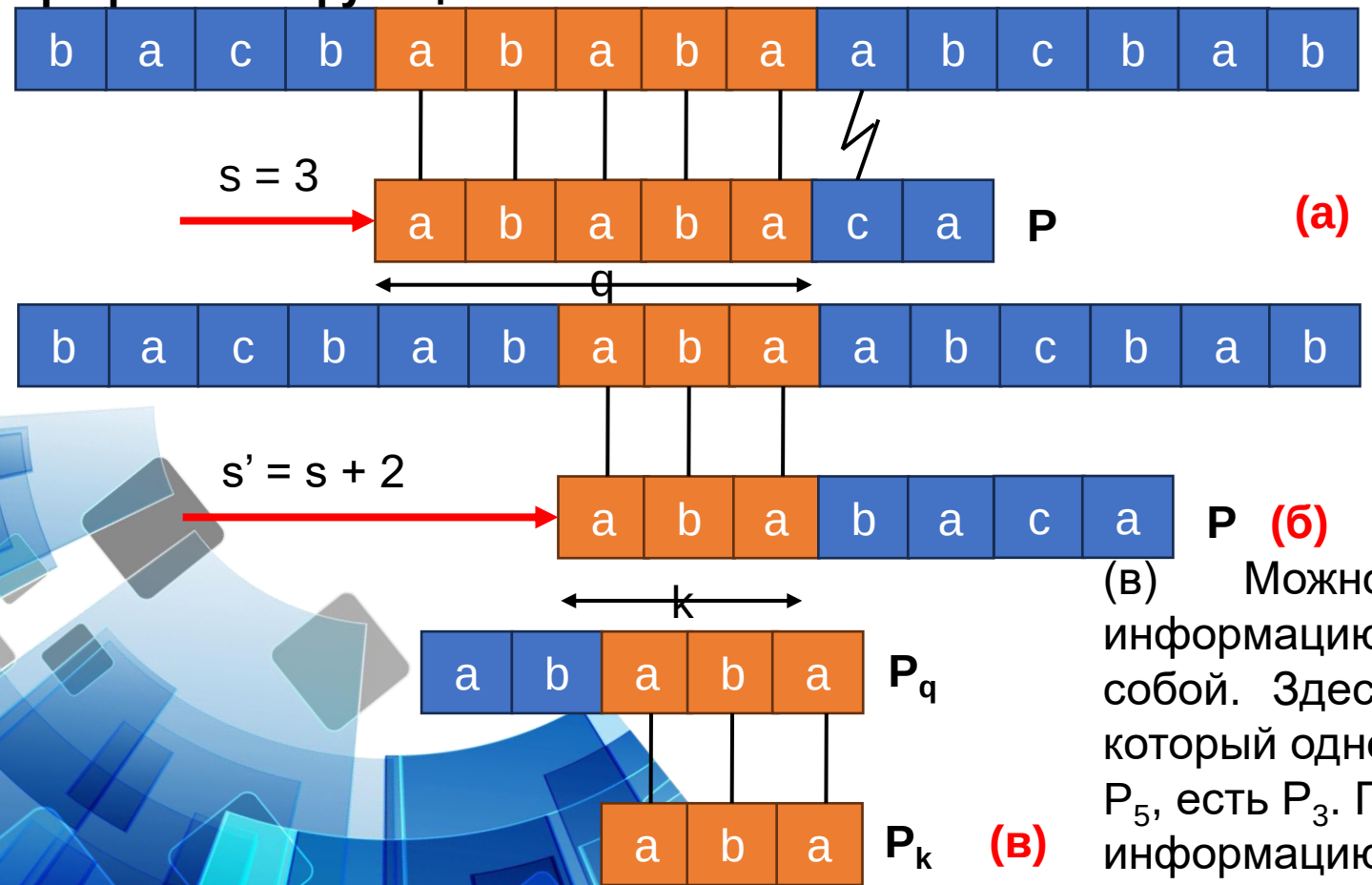
Алгоритм Кнута-Морриса-Пратта

Алгоритм сравнения строк, который был предложено Кнутом (Knuth), Моррисом (Morris) и Праттом (Pratt). В нем удается избежать вычисления функции переходов δ, а благодаря использованию вспомогательной функции π, которая вычисляется по заданному образцу за время Θ(m) и хранится в массиве π[1…m], время сравнения в этом алгоритме оказывается равным Θ(n). Массив τ позволяет эффективно (в амортизированном смысле) вычислять функцию δ "на лету", т.е. по мере необходимости. Грубо говоря, для любого состояния q = 0, 1, …,m и любого символа a ∈ Σ величина π[q] содержит информацию, необходимую для вычисления величины δ(q,a), но не зависит от a. Поскольку массив π содержит только m элементов (в то время как в массиве δ их O(m|Σ|), вычисляя на этапе предварительной обработки функцию π вместо функции δ, удается уменьшить время предварительной обработки образца в |Σ| раз.

Алгоритм Кнута-Морриса-Пратта

Префиксная функция π для некоторого образца инкапсулирует сведения о том, в какой мере образец совпадает сам с собой после сдвигов. Эта информация позволяет избежать ненужных проверок в простейшем алгоритме поиска подстрок и предвычисления функции δ при использовании конечных автоматов.

Префиксная функция π.



(а) Образец P = ababaca, сдвинутый относительно T так, что совпадают первые q = 5 символов. Совпадающие символы выделены серой штриховкой и соединены вертикальными линиями.

(б) Пользуясь одной лишь информацией о пяти совпадающих символах, можно вывести, что сдвиг s + 1 недопустим, но сдвиг s' = s + 2 согласуется со всем, что мы знаем о тексте, а значит, потенциально допустим.

(в) Можно предварительно вычислить полезную информацию для таких выводов, сравнив образец с самим собой. Здесь мы видим, что наидлиннейший префикс P, который одновременно является истинным суффиксом P_5, есть P_3. Представим эту предвычисленную информацию в массиве π, так что π[5] = 3.

Алгоритм Кнута-Морриса-Пратта

Известно, что символы P[1…q] образца соответствуют символам T[s + 1… s+ q] текста.
Каков наименьший сдвиг s' > s, такой, что для некоторого k < q
P[1…k] = T[s' + 1…s' + k] , (2.6)
где s' + k = s + q?

Другими словами, зная, что необходимо найти наидлиннейший истинный префикс P_k строки P_q, который одновременно является суффиксом T_{s+q}. (Поскольку s' + k = s + q, если заданы s и q, то поиск наименьшего сдвига s' эквивалентен поиску длины наибольшего префикса к.) Мы добавляем разность q - k длин этих префиксов P к сдвигу s, чтобы получить новый сдвиг s', что s' = s + (q - k). В лучшем случае k = 0, так что s' = s + q и мы тут же пропускаем сдвиги s + 1, s + 2, …, s + q - 1. В любом случае при новом сдвиге s' не нужно сравнивать первые k символов P с соответствующими символами T, поскольку (2.6) гарантирует, что они совпадают.

Необходимую информацию можно вычислить путем сравнения образца с самим собой. Поскольку T[s' + 1…s' + k] является частью известного фрагмента текста, она является суффиксом строки P_q. Поэтому уравнение (2.6) можно рассматривать как запрос о максимальном значении k < q, таком, что $P_k \supset P_q$. Тогда следующий потенциально допустимый сдвиг равен s' = s + (q - k). Оказывается, что удобнее хранить для каждого значения q количество k совпадающих при новом сдвиге s' символов, чем, например, величину s' - s.

Формализуем предварительно вычисляемую информацию следующим образом. Для заданного образца P[1…m] префиксной функцией для P является функция π: {1, 2, …,m} -> {0, 1, …,m - 1}, такая, что **$\pi[q] = \max\{k : k < q$ и $P_k \supset P_q\}$**

Алгоритм Кнута-Морриса-Пратта

```
// Функция для вычисления префикс-функции (Compute-Prefix-Function)
int *Compute_Prefix_Function(char *P) {
    int m = strlen(P);                                  // Длина образца
    int *pi = (int *)malloc((m + 1) * sizeof(int)); // Массив для префикс-функции (индекс с 1)
    pi[1] = 0;                                          // Первый символ не имеет префиксов
    int k = 0;                                          // Длина текущего префикса
    for (int q = 2; q <= m; q++) {
        while (k > 0 && P[k] != P[q - 1]) //Сравниваем символы (индексация с 0, поэтому P[q-1])
            k = pi[k]; // Переходим к меньшему префиксу
        if (P[k] == P[q - 1]) // Если символы совпадают
            k = k + 1; // Увеличиваем длину префикса
        pi[q] = k; // Сохраняем длину префикса для позиции q
    }
    return pi;
}
```

С помощью методов группового анализа можно показать, что время работы процедуры Compute-Prefix-Function равно Θ(m). Единственной тонкостью при этом является показ того, что всего цикл while выполняется O(m) раз. Общее количество итераций цикла while не превышает m - 1, и время работы процедуры Compute-Prefix-Function составляет Θ(m).

Алгоритм Кнута-Морриса-Пратта

```
// Функция KMP для поиска образца в тексте
void KMP_Matcher(char *T, char *P) {
    int n = strlen(T);                    // Длина текста
    int m = strlen(P);                    // Длина образца
    int *pi = Compute_Prefix_Function(P); // Вычисляем префикс-функцию
    int q = 0;                            // Количество совпадающих символов
    for (int i = 1; i <= n; i++) {        // Сканируем текст слева направо (индексация с 1)
        while (q > 0 && P[q] != T[i - 1]) // Пока q > 0 и символы не совпадают
            q = pi[q]; // Переходим к предыдущему возможному совпадению
        if (P[q] == T[i - 1]) // Если символы совпадают
            q = q + 1; // Увеличиваем количество совпадающих символов
        if (q == m) {                                                  // Если совпал весь образец
            printf("Образец находится со смещением %d\n", i - m); // Выводим позицию
            q = pi[q];                                                 // Ищем следующее вхождение
        }
    }
    free(pi); // Освобождае
}
```

Можно показать с помощью группового анализа, что время сравнения процедуры KMP-Mather равно Θ(n).
Благодаря использованию функции π вместо функции δ, которая используется в процедуре Finite-Automaton-Matcher, время предварительной обработки образца уменьшается с O(m|Σ|) до Θ(m), при том что время фактического сравнения остается ограниченным Θ(n).

Корректность вычислени префиксной функции

Положим
$\pi^*[q] = \{\pi[q], \pi^{(2)}[q], \pi^{(3)}[q], \ldots, \pi^{(t)}[q]\}$,
где $\pi^{(i)}[q]$ определена в терминах функциональной итерации, так что $\pi^{(0)}[q] = q$ и $\pi^{(i)}[q] = \pi[\pi^{(i-1)}[q]]$ для $i > 1$, и где последовательность $\pi^*[q]$ останавливается по достижении $\pi^{(t)}[q] = 0$.

Лемма 2.5 (Лемма об итерации префиксной функции)
Пусть P - образец длиной m с префиксной функцией π. Тогда для $q = 1, 2, \ldots, m$ имеем $\pi^*[q] = \{k : k < q \text{ и } P_k \supset P_q\}$.

Доказательство. Сначала докажем, что $\pi^*[q] \subseteq \{k : k < q \text{ и } P_k \supset P_q\}$ или, что эквивалентно,

из $i \in \pi^*[q]$ следует $P_i \supset P_q$ (2.7)

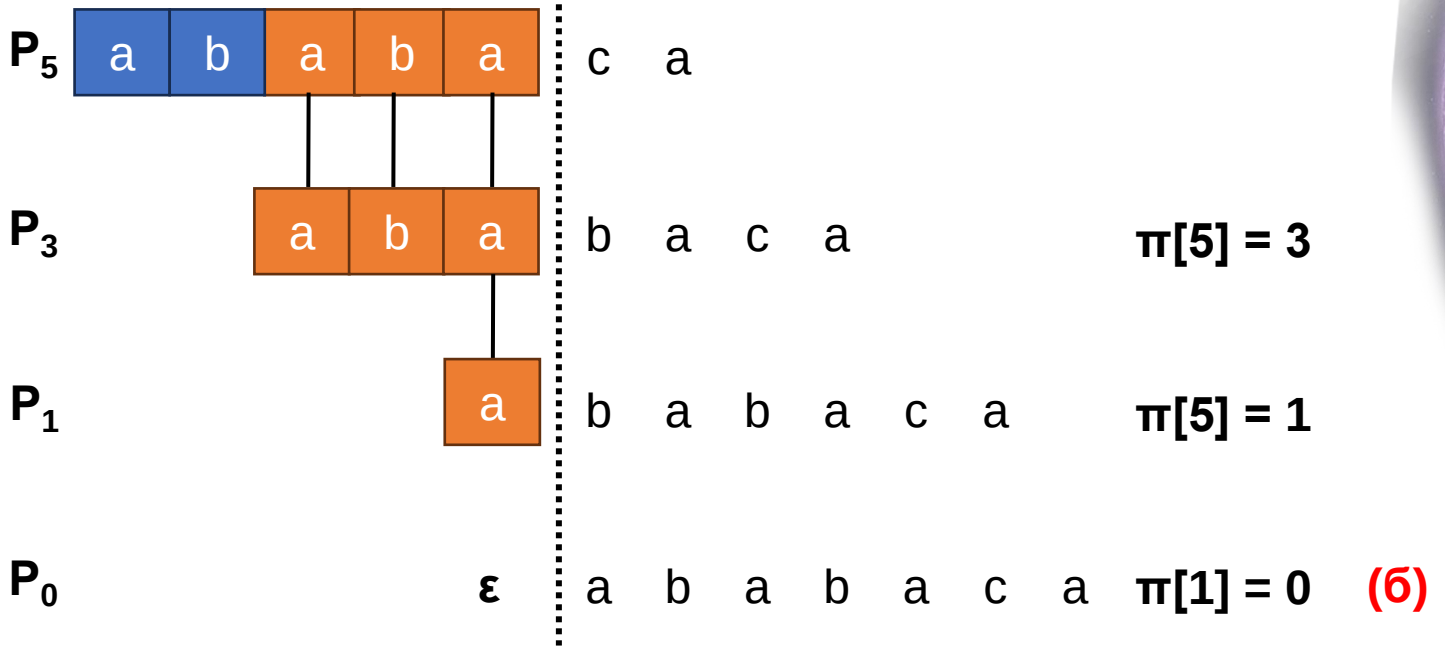
Если $i \in \pi^*[q]$, то $i = \pi^{(u)}[q]$ для некоторого $u > 0$. Докажем уравнение (2.7) по индукции по u. Для $u = 1$ имеем $i = \pi[q]$, и наше утверждение следует из того, что $i < q$ и $P_{\pi[q]} \supset P_q$ по определению π. Используя отношения $\pi[i] < i$ и $P_{\pi[i]} \supset P_i$ и транзитивность $<$ и $\supset$, устанавливаем справедливость утверждения для всех i из $\pi^*[q]$. Следовательно, $\pi^*[q] \subseteq \{k : k < q \text{ и } P_k \supset P_q\}$.

Теперь докажем, что $\{k : k < q \text{ и } P_k \supset P_q\} \subseteq \pi^*[q]$, методом от противного. Предположим, что множество $\{k : k < q \text{ и } P_k \supset P_q\} - \pi^*[q]$ непустое, и пусть j представляет собой наибольшее число этого множества. Поскольку $\pi[q]$ является наибольшим значением в $\{k : k < q \text{ и } P_k \supset P_q\}$ и $\pi[q] \in \pi^*[q]$, должно выполняться $j < \pi[q]$, так что пусть j' обозначает наименьшее целое из $\pi^*[q]$, большее, чем j. (Можно выбрать $j' = \pi[q]$, если никакие другие числа в $\pi^*[q]$ не превышают j.) Имеем $P_j \supset P_q$, поскольку $j \in \{k : k < q$и Pk Pq), и из $j' \in \pi^*[q]$ и уравнения (2.7) получаем $P_{j'} \supset P_q$. Таким образом, $P_j \supset P_{j'}$ согласно лемме 2.1, и j является наибольшим значением, меньшим j' и обладающим этим свойством. Следовательно, должны выполняться $\pi^*[j'] = j$ и, поскольку $j' \in \pi^*[q]$, $j \in \pi^*[q]$. Это противоречие и доказывает лемму.

Алгоритм Кнута-Морриса-Пратта

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
π[i]	0	0	1	2	3	0	1

(а)



(б)

Иллюстрация леммы 2.5 для образца P = ababaca и q = 5. (а) Функция π для указанного образца. Поскольку π[5] = 3, π[3] = 1 и π[1] = 0, итерируя π, получаем π*[5] = {3, 1, 0}.

(б) Мы смещаем образец P вправо и замечаем, когда некоторый префикс P_k образца P соответствует некоторому истинному суффиксу P_5; мы получаем соответствие при k = 3, 1 и 0. На рисунке в первой строке приведен образец P, а пунктирная вертикальная линия проведена после P_5. Последовательные строки показывают все сдвиги P, которые приводят к тому, что некоторый префикс P_k строки P соответствует некоторому суффиксу P_5. Совпадающие символы выделены штриховкой и соединены вертикальными линиями. Таким образом, $\{k: k < 5 \text{ и } P_k \sqsupset P_5\} = \{3,1,0\}$. Лемма 2.5 утверждает, что $\pi^*[5] = \{k : k < q \text{ и } P_k \sqsupset P_q\}$ для всех q.

Корректность вычисления префиксной функции

Лемма 2.6

Пусть P является образцом длиной и пусть π представляет собой префиксную функцию для P. Если для q = 1,2,…,m выполняется π[q] > 0, то π[q] - 1 ∈ π*[q - 1].

Доказательство. Пусть r = π[q] > 0, так что r < q и $P_r \supset P_q$; таким образом, r-1 < q-1 и $P_{r-1} \supset P_{q-1}$ (отбрасывая последние символы из P_r и P_q, что можно сделать, поскольку r > 0). Таким образом, согласно лемме 2.5 r - 1 ∈ π*[q - 1]. А значит, π[q] - 1 = r - 1 ∈ π*[q - 1].

Определим для q = 2,3, …, m подмножество $E_{q-1} \subseteq$ π*[q-1] как

E_{q-1} = {k ∈ π*[q - 1] : P[k + 1] = P[q]}

= {k : k < q - 1 и $P_k \supset P_{q-1}$ и P[k + 1] = P[q]} (из леммы 2.5)

= {k : k < q - 1 и $P_{k+1} \supset P_q$}

Множество E_{q-1} состоит из значений k < q - 1, для которых $P_k \supset P_{q-1}$ и для которых имеем $P_{k+1} \supset P_q$, поскольку P[k + 1] = P[q]. Таким образом, E_{q-1} состоит из значений k ∈ π*[q - 1], таких, что можно расширить P_k до P_{k+1} и получить истинный суффикс P_q.

Корректность вычисления префиксной функции

Следствие 1.7

Пусть P является образцом длиной m и пусть π представляет собой префиксную функцию для P. Для q = 2,3, …, m

$$\pi[q] = \begin{cases} 0, \text{если } E_{q-1} = \emptyset, \\ 1 + \max\{k \in E_{q-1}\}, \text{если } E_{q-1} \neq \emptyset. \end{cases}$$

Доказательство. Если E_{q-1} - пустое множество, не существует $k \in \pi^*[q-1]$ (включая k = 0), для которого можно расширить P_k до P_{k+1} и получить истинный префикс P_q. Следовательно, π[q] = 0.

Если множество E_{q-1} непустое, то для каждого $k \in E_{q-1}$ имеем $k + 1 < q$ и $P_{k+1} \sqsupset P_q$. Следовательно, из определения π[q] имеем

$$\pi[q] \geq 1 + \max\{k \in E_{q-1}\}. \quad (2.8)$$

Заметим, что π[q] > 0. Пусть r = π[q] - 1, так что r + 1 = π[q] и, следовательно, $P_{r+1} \sqsupset P_q$. Поскольку r + 1 > 0, имеем P[r + 1] = P[q]. Кроме того, согласно лемме 2.6 имеем $r \in \pi^*[q-1]$. Следовательно, $r \in E_{q-1}$, и, таким образом, $r \leq \max\{k \in E_{q-1}\}$, или, что эквивалентно,

$$\pi[q] \leq 1 + \max\{k \in E_{q-1}\}. \quad (2.9)$$

Объединение уравнений (2.8) и (2.9) завершает доказательство.

Корректность алгоритма Кнута-Морриса-Пратта

Процедуру KMP-Matcher можно рассматривать как видоизмененную реализацию процедуры Finite-Automaton-Matcher, использующую префиксную функцию π для вычисления переходов между состояниями.

Домашнее задание!

Адженда

Системные вызовы и ядра	Строковые алгоритмы	Vim – основы
35 минут	30 минут	25 минут

Что такое Vim?

Vim (Vi IMproved) — это текстовый редактор, созданный на основе Vi.

- Разработчик: Брам Моленар (Bram Moolenaar), первая версия вышла в 1991 году.
- Работает в терминале, доступен на большинстве систем (Linux, macOS, Windows).
- Основная идея: редактирование текста без мыши, только с клавиатуры.

Фото Брама Моленара

Почему Vim популярен?

- Скорость: Минимум движений рук, всё делается с клавиатуры.
- Универсальность: Работает на любом сервере или устройстве с терминалом.
- Настраиваемость: Плагины, скрипты, конфигурации через .vimrc.
- Мощные команды: Быстрое редактирование, поиск, замена.

Используется программистами, админами, писателями.

Основные + VIM

Не требуется графический интерфейс

- Может использоваться на серверах
- Требует мало ресурсов для работы

Для работы нужна только клавиатура

Удобство и широкие возможности

Огромное сообщество и плагины (например, NERDTree, YouCompleteMe).

Минусы:

Крутая кривая обучения (сложно для новичков).

Нет графического интерфейса (хотя есть gVim).

Требует привыкания к режимам.

Основные команды Vim

h, j, k, l — влево, вниз, вверх, вправо.
w — прыгнуть к следующему слову, b — к предыдущему.
Редактирование:
i — войти в режим вставки перед курсором.
dd — удалить строку, yy — скопировать строку, p — вставить.
Сохранение и выход:
:w — сохранить, :q — выйти, :wq — сохранить и выйти.
Поиск и замена:
/text — искать "text", :%s/old/new/g — заменить "old" на "new".
Открываем файл: vim example.txt
Входим в режим вставки: i
Пишем текст: "Привет, я учу Vim!"
Выходим из режима вставки: Esc
Сохраняем и выходим: :wq
Бонус: Удаляем строку с ошибкой: dd, копируем другую: yy, вставляем: p.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Функция для вычисления префикс-функции (Compute-Prefix-Function)
int *Compute_Prefix_Function(char *P)
{
    int m = strlen(P);                                  // Длина образца
    int *pi = (int *)malloc((m + 1) * sizeof(int)); // Массив для префикс-функции (индексация с 1)
    pi[1] = 0;                                          // Первый символ не имеет префиксов
    int k = 0;                                          // Длина текущего префикса

    for (int q = 2; q <= m; q++)
    {
        while (k > 0 && P[k] != P[q - 1])
```