Программа, подсчитывающая количество символов латиницы и кириллицы

Пояснительная записка

Исполнитель:

Студент группы БПИ193-1 ФКН НИУ ВШЭ

Гриценко Егор Андреевич

“17” октрября 2020 г.

Москва 2020

1. Введение

Программа выполнена в рамках первого микропроекта дисциплины “Архитектура вычислительных систем” второго курса бакалавриата программной инженерии ФКН НИУ ВШЭ.

2. Исходное условие задания микропроекта

Разработать программу, вычисляющую число букв латиницы и кириллицы в заданной ASCII-строке.

3. Описание допустимой среды выполнения программы и средств разработки

Программа может выполняться на ОС архитектуры AMD64, основанных на GNU/Linux, с таблицей syscall’ов ядра сооветствующей таковой у ядер Linux версии 5.8.7. Программа непосредственно взаимодействует с ядром и не требует наличие libc.

Программа написана на GNU диалекте AT&T ASM; сборка программмы осуществляется посредством GNU Assembler (as) и GNU Linker (ld). Функции программы соблюдают конвенцию вызовов System V, используемую в AMD64 Linux.

Среда разработки и тестирования: CPU Intel(R) Core(TM) i7-8565U, ОС Arch Linux (версия ядра 5.8.7), локаль en_us.ut8.

4. Отличия от условия, продиктованные выбранной средой

В условии указано использовать ASCII строки. Однако на современных системах (особенно в Linux) для представления символов, не входящих в стандартный набор ASCII, используется UTF8. Поскольку локаль на системе (в частности, в терминале проверяющего) скорее всего также будет UTF8, разработанная программа обрабатывает именно UTF8 строки; при этом латиницей и кириллицей считаются только те символы латиницы и русской кириллицы, которые входят в ASCII 866.

Также в условии указано использовать cdelc. В AMD64 Linux используется конвенция вызовов, описанная в System V AMD64 ABI, поэтому вместо cdelc используется именно она.

5. Организация ввода-вывода

Интерфейс ввода было решено организовать схожим с cat образом. Программа производит попытки чтения из стандартного ввода до тех пор, пока не достигается конец потока, определяемый нулевым количеством прочитанных байт.

Чтение стандартного ввода осуществляется посредством syscall’a read в буфер. Буфер хранится на стэке, его размер определяется в начале выполнения программы как половина мягкого лимита на размер стэка. Этот лимит получается syscall’ом getrlimit. Как следствие, если программа вызвана форком, исчерпавшим стэк чуть более, чем наполовину, произойдёт segmentation fault. Такой случай, правда, маловероятен. (Если специально не стараться, при ручном запуске из терминала segfault’a не будет.)

Решение использовать стэк было принято, поскольку было интересно попытаться избежать резервирование нового сегмента памяти и создания его отображения посредством syscall’a

mmap, и вместо этого воспользоваться уже существующим отображением для stack’a, которое иначе было бы почти неиспользуемым.

Изначально было принято решение отвести весь стэк (за исключением приблизительно 256 байт необходимых для хранения регистров, адресов возврата и прочих мелких расходов) под буфер, однако анализ документации и релевантных форумов не выявил быстрый и элегантный способ получения основания стэка в Linux. Предлагаемые решения основывались либо на последовательном подъёме по памяти от %rsp пока не будет получен SIGSEGV, либо же на parsing’e /proc/self/maps.

Программа выводит информацию о количестве прочитанных букв в стандартный вывод. Вывод таблицы со статистикой производится построчно: для каждой строки таблицы сначала в том же, что и для ввода, буфере формируется её представление, затем она записывается в стандартный вывод посредством syscall’a write.

6. Организация представления числа в виде строки из цифр

Для преобразования числа в строку из десятичных цифр написана функция char *itoa(long number, char *buffer), записывающая в буфер представление этого числа и возвращающая указатель на следующий за последним байтом этого представления байт.

Определение десятичных цифр производится от младших разрядов ко старшим: каждая цифра добавляется в стэк. Далее цифры копируются из стэка в буфер.

7. Организация parser’a

В UTF8 символы имеют переменную длину, но не более 4-х байт. Длина определяется позицией первого слева нуля: если его позиция 0, то символ является однобайтовым ASCII символом, иначе позиция нуля равна количеству байт в символе. Кириллические русские символы являются двухбайтовыми и начинаются только с $0xd0 или с $0xd1 соответственно. Все символы английской латиницы являются ASCII символами.

Примерный алгоритм parser’a стандартного ввода:

1. Загрузить предыдущее состояние в буфер: байты сложного символа, который на предыдущей итерации оказался разорван границами буфера.
2. Заполнить буфер данными из стандартного ввода.
3. Если ничего не было прочитано, выход.
4. Загрузить текущий байт и найти в нём позицию первого слева 0 (использован bsr).
5. Если та позиция равна 0, то это ASCII-символ, увеличиваем количество встреченных символов, равных текущему, на 1, и перейти к parsing’у следующего символа.
6. Иначе, символ возможно русский.

6.1.1. Если текущий байт --- 0xd0 или 0xd1, то он, возможно, русский, установить количество “ожидаемых” байт в 1.

6.1.2. Если все байты буфера уже прочитаны, выход из цикла. Иначе, прочитать второй байт, закодировать символ числом от 0 до 256 и увеличить количество встреченных символов с таким кодом.

6.1.3. Установить число “ожидаемых” байт в 0. Перейти к parsing’у следующего символа.

6.2.1. Если же текущий символ не русский, установить число “ожидаемых” байт сообразно позиции первого слева 0 в первом байте и увеличить указатель на текущий байт на эту позицию. Перейти к парсингу следующего символа.

7. Конец цикла. Если чтение буфера завершено, но есть недополученные байты, на основании их числа сохранить все первые байты разорванного границами буфера недопрочитанного символа как следующее начальное состояние парсера.

8. Выход. Если хоть какие-то байты были прочитаны, вернуть положительное число, иначе вернуть 0.

Следует отметить, что для ASCII-символов и потенциально русских символов инкремент счётчиков производится без проверки на строгую принадлежность к латинице и кириллице. Parser сделан с надеждой уменьшить число условных переходов, в разы замедляющих работу parser’a; таким образом была произведена попытка достичь выигрыш в производительности за счёт пары килобайт памяти.

Массивы счётчиков обозначины как ascii_couner и как utf8_c_counter соответственно.

8. Тестирование

Ниже приведена часть тестов, на которых проверялась работа программы.

1. В терминале вызвать программу, ввести данные, содержащиеся в файле 0t. Проверить соответствие результата входным данным.

2. В терминале вызвать программу, ввести данные, содержащиеся в файле 1t или в директории с программой выполнить “./program_name < path_to_1t_here”. Проверить соответствие результата входным данным.

3. В терминале несколько раз выполнить “curl -s https://cs.hse.ru | program_name”. Разнящиеся результаты могут означать ошибки curl, ошибки интернет соединений, быстрое измение контента на https://cs.hse.ru, а также неправильную работу программы на данных превышающих выделяемый буфер.

4. Сгенерировать в оперативной памяти случайным образом заполненный файл объёмом 8 GiB (Например “dd if=/dev/urandom bs=32M count=256 of=/tmp/8”). В директории с программой выполнить (dd if=/tmp/8 | ./program_name).

Для 4-ого теста отдельно отметим, что фиксируемая dd скорость обработки данных при тестировании колебалась в пределах 250---260 MiB/sec вне зависти от того, использовались ли для работы с буфером “цепочные” lods инструкции или нет. Возможно, это вызвано большим количеством условных переходов в цикле parser’a: для сравнения, подсчёт количества различных байт в случайном массиве происходит со скоростью не менее 2-х GiB/sec на той же машине. Возможно, это вызвано тем, что современные AMD64 процессоры являются “RISC-внутри”, и декодируют mov (%rsi), %al ; inc %rsi и lodbs в тот же набор внутренних инструкций.

Скриншоты тестов прилагаются в репозитории с исходным кодом ПО.

9. Компиляция

Компиляция была произведена как:

as task3.s -o task3.o

ld task3.o -o task3

10. Список некоторых источников, использованных при разработке

1. Using as // https://sourceware.org/binutils/docs/as/index.html
2. Linux man pages: section 2 // https://man7.org/linux/man-pages/dir_section_2.html
3. System V Application Binary Interface AMD64 Architecture Processor Supplement Draft Version 0.99.6 // https://refspecs.linuxfoundation.org/elf/x86_64-abi-0.99.pdf
4. x86 and amd64 instruction reference // https://www.felixcloutier.com/x86/
5. Searchable Linux Syscall Table for x86 and x86_64 // https://filippo.io/linux-syscall-table/
6. Elixir Cross Referencer - Explore source code in your browser - Particularly useful for the Linux kernel and other low-level projects in C/C++ (bootloaders, C libraries…) // https://elixir.bootlin.com/linux/latest/source
7. Linux kernel // https://github.com/torvalds/linux
8. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs // https://www.agner.org/optimize/instruction_tables.pdf
9. ASCII Table and Description // http://www.asciitable.com/
10. UTF-8 // https://en.wikipedia.org/wiki/UTF-8
11. UTF-8 encoding table and Unicode characters page with code points U+0400 to U+04FF // https://www.utf8-chartable.de/unicode-utf8-table.pl?start=1024