AXIOM JDK

Axiom Linux
Руководство по реализациям libc

Axiom Linux I Версия 2.0 I Декабрь 2025

Программное обеспечение АКСИОМ содержит программное обеспечение с открытым исходным кодом. Дополнительная информация о коде сторонних разработчиков доступна на сайте https://axiomjdk.ru/third_party_licenses. Для дополнительной информации о том, как получить копию исходного кода, можно обратиться по адресу info@axiomjdk.ru.

ДАННАЯ ИНФОРМАЦИЯ МОЖЕТ ИЗМЕНЯТЬСЯ БЕЗ ПРЕДВАРИТЕЛЬНОГО УВЕДОМЛЕНИЯ. АКСИОМ ПРЕДОСТАВЛЯЕТ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ "КАК ЕСТЬ" БЕЗ КАКИХ-ЛИБО ГАРАНТИЙ, АКСИОМ ПРЯМО ОТКАЗЫВАЕТСЯ ОТ ВСЕХ ПОДРАЗУМЕВАЕМЫХ ГАРАНТИЙ, ВКЛЮЧАЯ, НО НЕ ОГРАНИЧИВАЯСЬ ПОДРАЗУМЕВАЕМЫМИ ГАРАНТИЯМИ ТОВАРНОЙ ПРИГОДНОСТИ И ПРИГОДНОСТИ ДЛЯ ОПРЕДЕЛЕННОЙ ЦЕЛИ.

АКСИОМ НИ ПРИ КАКИХ ОБСТОЯТЕЛЬСТВАХ НЕ НЕСЕТ ОТВЕТСТВЕННОСТИ ЗА ЛЮБЫЕ КОСВЕННЫЕ, СЛУЧАЙНЫЕ, СПЕЦИАЛЬНЫЕ, ШТРАФНЫЕ ИЛИ КОСВЕННЫЕ УБЫТКИ, ИЛИ УБЫТКИ ОТ ПОТЕРИ ПРИБЫЛИ, ДОХОДА, ДАННЫХ ИЛИ ИСПОЛЬЗОВАНИЯ ДАННЫХ, ПОНЕСЕННЫЕ ВАМИ ИЛИ ЛЮБОЙ ТРЕТЬЕЙ СТОРОНОЙ, БУДЬ ТО В РЕЗУЛЬТАТЕ ДЕЙСТВИЯ ДОГОВОРА ИЛИ ДЕЛИКТА, ДАЖЕ ЕСЛИ АКСИОМ БЫЛО ПРЕДУПРЕЖДЕНО О ВОЗМОЖНОСТИ ТАКИХ УБЫТКОВ.

Использование любого программного продукта АКСИОМ регулируется соответствующим лицензионным соглашением, которое никоим образом не изменяется условиями данного уведомления. Программные продукты и фирменные наименования: Axiom JDK, Axiom JDK Pro, Axiom Runtime Container Pro, Axiom Linux, Libercat, Libercat Certified и АКСИОМ принадлежат АКСИОМ и их использование допускается только с разрешения правообладателя.

Товарный знак Linux® используется в соответствии с сублицензией от Linux Foundation, эксклюзивного лицензиата Линуса Торвальдса, владельца знака на всемирной основе. Java и OpenJDK являются товарными знаками или зарегистрированными товарными знаками компании Oracle и/или ее аффилированных лиц. Другие торговые марки являются собственностью их соответствующих владельцев и используются только в целях идентификации.

Содержание

1. Введение

Axiom Linux имеет два варианта "libc", основанные на двух разных реализациях "libc", а именно "glibc" и "musl". "musl" дополнительно представлен двумя вариантами, такими как "musl-default" и "musl-perf", и его можно легко поменять в установленной системе с помощью менеджера пакетов apk.

> **Примечание:**
>
> Когда в тексте ниже упоминается "musl", подразумеваются оба варианта "musl".

В этом документе представлен общий сравнительный анализ наиболее важных различий в реализациях "libc". Для получения дополнительной информации см. следующие документы:

- Functional differences from "glibc"
- Comparison of C/POSIX standard library implementations for Linux

2. Локали

Оба варианта "libc" по умолчанию используют локаль "C.UTF-8". В "musl" она встроена, а в "glibc" генерируется на уровне пакета. Изменить локаль в "musl" сложно, и поддержка переменной среды `MUSL_LOCPATH` по-прежнему ограничена.

Обычно функции, которых нет в самой библиотеке "musl", находятся в сторонних проектах, включая локали.

Напротив, "glibc" имеет впечатляющий набор локалей, которые сразу доступны в пакете `glibc-locales`.

3. DNS

В отличие от "glibc", "musl" выполняет параллельные запросы к серверам имен, найденным в /etc/resolv.conf, и возвращает только первый принятый ответ. В "glibc" запросы выполняются последовательно, то есть следующий сервер запрашивается только после истечения времени ожидания предыдущего. Обе версии ограничены 3 серверами имен.

Параллельные запросы обеспечивают лучшую производительность в "musl", но в некоторых случаях это может быть неприемлемо, и в этом случае на помощь может прийти кэширование локального DNS-сервера. Кэширование снижает нагрузку на процессор и сеть и может даже ускорить разрешение имен в "glibc", чтобы преодолеть последовательный характер запросов.

Рассмотрим простую настройку сервера кэширования с использованием гибкого инструмента, такого как dnsmasq, следующим образом.

Настройка кэширующего DNS-сервера

```
apk add dnsmasq

cat > /etc/resolv.conf << EOF
127.0.0.1
EOF

cat >> /etc/dnsmasq.conf << EOF
port=53
listen-address=127.0.0.1
strict-order
no-resolv
no-poll
server=IP_address_1
server=IP_address_2
EOF

rc-update add dnsmasq default
rc-service dnsmasq start
```

DNS TCP

`musl` не поддерживал TCP до версии 1.2.3-r15, но теперь, если получен ответ UDP с флагом

`Truncated`, `musl` отправляет новый запрос по TCP.

Директивы DNS в musl /etc/resolv.conf

Директива `search` доступна в `musl` начиная с версии 1.1.13, следовательно, Axiom Linux поддерживает ее с самого начала. Директива `domain` (устаревшая) ведет себя идентично директиве `search`, хотя предполагалось, что в ней должна быть только одна запись.

Открытые "параметры" в "musl`:

- `ndots:n`: значение по умолчанию равно 1, что означает, что если в имени есть какие-либо точки, список поиска не используется.
- `timeout:n`: Количество времени, в течение которого распознаватель ожидает ответа. Интервал повторных попыток рассчитывается как "тайм-аут / попытки". Значение по умолчанию равно 5 секундам.
- `attempts:n`: Количество попыток. Значение по умолчанию равно 2.

Значения по умолчанию для вышеуказанных параметров очень похожи на `glibc`.

4. Производительность

`musl` - это облегченная библиотека, которая намного меньше, чем "glibc". В ней нет перегруженных или сложных функций, управления версиями и чрезмерного использования malloc.

Образ Docker stream-glibc	**Образ stream-musl с musl-perf**
8.4 MB	3.33MB

Потребление ресурсов также невелико, например, размер стека по умолчанию в `musl` составляет всего 128 КБ, в отличие от значения по умолчанию в `glibc`. Это может обеспечить дополнительные преимущества для приложений с большим количеством потоков. При необходимости `musl` поддерживает изменение размера стека с помощью `pthread_attr_setstacksize` или при связывании с параметром `-wl,-z,stack-size=N`. Размер может быть увеличен или уменьшен.

malloc

Говоря о производительности, мы должны упомянуть "musl-perf". `musl-perf" теперь включает в себя "mimalloc", который обеспечивает преимущества "musl-default" (например, уменьшенную площадь атаки и меньший размер) при одновременном устранении недостатков в производительности, что приводит к повышению производительности системы. Для получения дополнительной информации см. Функции выделения памяти (memory allocators).

memcpy, memmove, memset и другие функции

Некоторые считают, что "glibc" быстрее, чем "musl" из-за оптимизированных строковых функций для различных функций процессора, таких как avx2, avx512 и т.д. Теперь такие оптимизации доступны в пакете "musl-perf", что отличает его от обычной реализации "musl".

Наилучшая реализация строковой функции выбирается во время выполнения в зависимости от возможностей центрального процессора. То же самое, что и в `glibc`. Хотя наличие нескольких реализаций функций увеличивает размер дистрибутива.

Давайте посмотрим на результаты тестов потокового копирования Phoronix на компьютере с поддержкой VX2. Это всего лишь пример теста, который использует только функцию `memcpy()`.

Дистрибутив	**мегабайт/с**
Axiom Linux (glibc)	21601
Axiom Linux (musl-default)	19023
Axiom Linux (musl-perf)	21663
Alpine 3.17	19184
Centos 9	21558
Red-Hat 8	20409
Debian 11	20509

Производительность "musl-perf" практически такая же, как в варианте Axiom Linux "glibc" или в других дистрибутивах на основе "glibc".

5. Функции выделения памяти (memory allocators)

`glibc` использует функцию `ptmalloc2` (threads malloc) для динамического выделения памяти общего назначения. Это одна из наиболее проверенных и быстрых реализаций malloc для нескольких потоков без конфликта блокировок. Она имеет хороший баланс скорости и памяти и настраиваемые параметры с помощью переменной окружения `GLIBC_TUNABLES`.

В `musl-default` есть функция `mallocng`, введенная и используемая по умолчанию начиная с версии 1.2.1 в 2020 году. Она известна своей усиленной защитой от переполнений буфера, ошибок использования памяти и лучшего предотвращения фрагментации.

Хотя musl-default известен своей компактностью и преимуществами в области безопасности, его реализация malloc по умолчанию может стать узким местом для приложений, для которых критически важна производительность. `musl-perf` в Axiom Linux использует `mimalloc` в качестве средства выделения памяти по умолчанию в musl libc. Этот выбор сделан для устранения ограничений производительности, часто связанных с реализацией malloc по умолчанию в musl-default, что может привести к значительному замедлению работы в некоторых приложениях, особенно в приложениях с большим объемом выделенной памяти. `mimalloc", разработанный Microsoft, представляет собой высокопроизводительный распределитель памяти, разработанный для того, чтобы быть быстрым и эффективным, и часто превосходящий в тестах другие ведущие распределители, такие как "jemalloc" и "tcmalloc".

Поскольку "mimalloc" интегрирован и используется по умолчанию в "musl-perf`, вам не нужно устанавливать его отдельно. Если вы хотите проверить различия между распределителем, встроенным в "musl-perf", и настройками по умолчанию в "mimalloc" и "mimalloc-secure", вы можете установить оба распределителя. Основные различия заключаются в следующем:

- Встроенный распределитель `musl-perf`:
    - Уровень защищенного режима mimalloc установлен на 1, что обеспечивает необходимую защиту от распространенных уязвимостей, связанных с кучей (heap), при сохранении хорошей производительности.
    - Экспортирован интерфейс "mi_collect(bool force)" для выполнения локального сжатия потоков кучи по требованию.
    - Функция Transparent Huge Pages (THP) по умолчанию отключена, чтобы снизить нагрузку на память, аналогично стандартному распределителю musl. При необходимости его все еще можно включить с помощью явной опции "MADV_HUGEPAGE" для больших выделений.

- mimalloc - THP включен, уровень защищенного режима установлен на 0
- mimalloc-secure - THP включен, уровень защищенного режима установлен на 4

В дополнение к встроенным функциям, существует ряд других внешних функций выделения динамической памяти, которые доступны как для "glibc", так и для `musl`:

- **mimalloc**: превосходит по скорости другие ведущие функции выделения памяти (jemalloc, tcmalloc и т.д.) и часто использует меньше памяти.
- **mimalloc-secure**: это mimalloc со встроенным защищенным режимом, который добавляет защитные страницы, рандомизированное выделение и зашифрованные свободные списки для защиты от различных уязвимостей. Хотя он имеет сниженную производительность по сравнению с mimalloc в среднем на 10% в различных тестах.
- **jemalloc**: реализация, в которой особое внимание уделяется предотвращению фрагментации и поддержке масштабируемого параллелизма.
- **rpmalloc**: функция выделения памяти общего назначения с кэшированием потоков без блокировок.

Чтобы переключиться на одну из этих функций, например на mimalloc, можно запустить следующую команду в Axiom Linux:

```
apk add mimalloc-global
```

Убедитесь, что он используется глобально:

```
ldd /bin/busybox
  /lib/ld-musl-x86_64.so.1 (0x7f2725f02000)
  /lib/libmimalloc.so.1.7 => /lib/libmimalloc.so.1.7 (0x7f2725e04000)
  libc.musl-x86_64.so.1 => /lib/ld-musl-x86_64.so.1 (0x7f2725f02000)
```

Для получения дополнительной информации см следующие документы:

- Glibc: malloc internals
- Glibc: tunables
- Musl: Mallocng algorithm high-level overview (ML)
- Musl: documentation for mallocng (ML)

6. Косвенная функция (ifunc)

Как `glibc`, так и `musl` поддерживают ifunc. Изначально `musl-default` не поддерживал его, но начиная с версии 1.2.3-r12 поддержка ifunc была добавлена и в Axiom Linux. Это связано с тем, что в пакеты была добавлена функция управления несколькими версиями функции GCC, например, в пакет gzip, который начиная с версии `gzip-1.12-r2` для работы требует поддержки ifunc.

7. Формат DT_RELR

Новый формат позволяет оптимально кодировать перемещения `R_*_RELATIVE` в общих объектах и размещать независимые исполняемые файлы (PIE), экономя размер результирующего двоичного файла.

Только "musl" поддерживает эту функцию начиная с версии 1.2.3-r9 в Axiom Linux. Вариант "glibc" с текущей версией 2.34 пока не обеспечивает такой поддержки.

Чтобы включить компактные относительные перемещения для ваших сборок в среде Axiom Linux musl, просто передайте линкеру флаг `-Wl,-z,pack-relative-relocs`. В результате вы получаете новый раздел перемещения ".rel.dyn" и экономите на размере созданного двоичного файла. В некоторых случаях уменьшение может достигать 5%.

Рассмотрим busybox, приложение, которое имеет более тысячи относительных перемещений:

busybox	**RELASZ**	**RELACOUNT**	**RELRSZ**	**Total Size**
RELA	25896	1057	-	833 200
with RELR	528 (-79%)	-	200	808 696 (-2.9%)

Использование REAL приводит к уменьшению размера секции перемещения на 79% и примерно на 2,9% общего размера.

Для получения дополнительной информации, см. Относительные перемещения и RELR.

8. Безопасность

Простота и меньший размер "musl" упрощают проверку его механизма и, что более важно, снижают вероятность сбоя, уменьшая площадь атаки.

Включение FORTIFY_SOURCE

Макрос `FORTIFY_SOURCE` помогает обнаруживать переполнения буфера в различных функциях, которые манипулируют памятью и строками в "libc", таких, как `memcpy`, `strcpy` и т.д., обеспечивая дополнительный уровень проверки для таких функций, которые потенциально являются источником ошибок переполнения буфера.

Чтобы включить его, установите значение выше 0, например `-D_FORTIFY_SOURCE=2` и включите оптимизацию компилятора с помощью флага `-O`.

`glibc` поддерживает его изначально, но `musl` требует заголовочных файлов из отдельного пакета. Чтобы убедиться, что скомпилированный двоичный файл собран с включенными дополнительными средствами защиты, сначала установите пакет `fortify-headers`, если он еще не установлен, следующим образом:

> **Примечание:**
>
> Пакет `fortify-headers` устанавливается автоматически, если пакет `build-base` установлен на `musl`. Таким образом, он может уже присутствовать на вашем компьютере.

```
apk add fortify-headers
```

Вы можете ознакомиться с тем, что входит в комплект поставки:

```
apk info -L fortify-headers
fortify-headers-1.1-r3 contains:
usr/include/fortify/fortify-headers.h
usr/include/fortify/poll.h
usr/include/fortify/stdio.h
usr/include/fortify/stdlib.h
usr/include/fortify/string.h
usr/include/fortify/strings.h
usr/include/fortify/unistd.h
usr/include/fortify/wchar.h
```

```
usr/include/fortify/sys/select.h
usr/include/fortify/sys/socket.h
```

Теперь вы можете скомпилировать приложения с `-D_FORTIFY_SOURCE=2`. `gcc` и `clang` по умолчанию ищут заголовочные файлы по пути `usr/include/fortify в `musl`. Следовательно, нет необходимости дополнительно указывать этот путь компиляторам.

К сожалению, вам нужно дизассемблировать двоичный файл, чтобы убедиться, что FORTIFY_SOURCE используется в `musl`. В `glibc` вы можете проверить символы `*_chk` в соответствующих функциях.

Axiom Linux
Руководство по реализациям libc