AXIOM JDK

Axiom Linux
Руководство по реализациям malloc

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. Реализация malloc в libc

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".

"glibc" использует функцию `ptmalloc2` (threads malloc) для динамического распределения памяти. Это одна из наиболее проверенных и быстрых реализаций malloc для нескольких потоков без конфликта блокировок. Она имеет хороший баланс скорости и занимаемой 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 памяти, 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 памяти `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

Для получения дополнительной информации см. следующие документы:

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

2. Внешние функции для распределения памяти

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. Правда 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.

3. Глобальное включение внешних функций

Чтобы переключиться на одну из внешних функций, например на `mimalloc`, мы можем запустить следующую команду в Axiom Linux:

```
apk add mimalloc-global
```

Пакеты `-global` работают с переменной окружения LD_PRELOAD, которая экспортируется через скрипты конфигурации `/etc/profile.d/`. Одновременно может быть установлен только один пакет `-global`. При установке нового пакета, он заменяет установленный пакет.

```
apk add jemalloc-global
...
(1/3) Installing jemalloc (5.3.0-r1)
(2/3) Installing jemalloc-global (5.3.0-r1)
Executing jemalloc-global-5.3.0-r1.post-install
*
* Please logout and login to apply the changes to your environment
* or exec 'source /etc/profile.d/jemalloc.sh'
*
(3/3) Purging mimalloc-global (1.7.7-r1)
```

Убедитесь, что он используется глобально:

```
ldd /bin/busybox
  ldd /bin/busybox
  /lib/ld-musl-x86_64.so.1 (0x7ff28dffc000)
  /lib/libjemalloc.so.2 => /lib/libjemalloc.so.2 (0x7ff28dc00000)
  ...
```

4. Индивидуальное включение внешних функций

Если вы хотите использовать внешние функции распределения памяти только для определенных пакетов, и более того, если вы хотите использовать разные внешние функции одновременно, вам не следует использовать глобальные пакеты. Вместо этого вручную укажите переменную окружения LD_PRELOAD только для определенных приложений.

- Установка нескольких функций распределения памяти:

```
apk add jemalloc mimalloc
(1/2) Installing jemalloc (5.3.0-r1)
(2/2) Installing mimalloc (1.8.1-r0)
Executing busybox-1.36.0-r7.trigger
OK: 866 MiB in 224 packages
```

- Использование jemalloc только для java приложения и mimalloc для python приложения:

```
jemalloc.sh java app
mimalloc.sh python -m app
```

Если вы не хотите использовать LD_PRELOAD и точно знаете, что приложение должно использовать определенные внешние функции распределения памяти прямо из коробки, создайте свое приложение статически с помощью одной из этих реализаций. Для этого Axiom Linux предоставляет пакеты `-dev` и `-static`.

5. Отладка glibc malloc

Все функции отладки в `glibc malloc` перенесены в отдельную библиотеку с именем `libc_malloc_debug.so.0` для повышения безопасности и производительности. Выполните следующую команду, чтобы включить ее:

```
export GLIBC_TUNABLES=glibc.malloc.check=3
LD_PRELOAD=/usr/lib/libc_malloc_debug.so.0 ./app
```

Для получения дополнительной информации смотрите: Обеспечение безопасности malloc в glibc.

6. Выбор функций распределения памяти

Выделение памяти в Java

JVM имеет свою собственную область памяти, которая поддерживается сборщиком мусора. Однако она также использует вызовы `malloc()` для реализации некоторых Java-объектов, таких, как:

- Буферы для процедур сжатия данных, которые представляют собой пространство памяти, необходимое библиотекам классов Java для чтения или записи сжатых данных, таких как файлы .zip или .jar.
- Распределение malloc с помощью JNI-кода приложения.
- Скомпилированный код, созданный JIT-компилятором.
- Потоки для сопоставления с потоками Java.

Если ваше приложение активно использует такие объекты, стоит посмотреть и проверить производительность с помощью альтернативных функций распределения памяти.

Выделение памяти в Python

Функция распределения памяти по умолчанию в Python — pymalloc, оптимизированный для небольших объектов (меньше или равных 512 байтам) с коротким временем жизни. Для более крупных объектов он возвращается к системной функции. Внутри он может использовать `mmap()` /`munmap()` и `malloc()`/`free()`. Вы можете отключить `pymalloc` с помощью переменной окружения `PYTHONMALLOC=malloc`. Возможно, будет полезно просмотреть различную статистику `pymalloc`, используя переменную `PYTHONMALLOCSTATS=1`.

Даже если у вас по умолчанию используется pymalloc в Python, вы можете оптимизировать свое приложение на Python с помощью другой функции выделения памяти.

Функции с лучшей производительностью

Давайте создадим пример приложения на Python. Мы не будем отключать pymalloc, а только проверим как работает malloc в libc, количество вызовов malloc и выделенный размер. Это может

показать что именно приложение делает во время выполнения, имеет ли смысл изменять или экспериментировать с другой функцией распределения памяти.

1. Сначала установите инструмент perf и python3:

```
apk add perf python3
```

2. Давайте создадим искусственный мини-тест, как показано ниже, который сможет читать один и тот же файл построчно (для простоты файл `/proc/kallsyms`) параллельно, используя 10 потоков чтения.

   По сути, мы сравниваем время, необходимое для создания потоков Python, и время, необходимое для чтения всего файла для всех потоков чтения.

```
import threading
import datetime


OUTER_THREADS_N = 2
INNER_THREADS_N = 5

def read_file():
    with open('/proc/kallsyms') as file:
        for line in file:
            pass

def spawn_threads(n, fn_name, args):
    threads = []
    for i in range(n):
        t = threading.Thread(target=fn_name, args=args)
        t.start()
        threads.append(t)
    for t in threads:
        t.join()

dt0 = datetime.datetime.now()
spawn_threads(OUTER_THREADS_N, spawn_threads, (INNER_THREADS_N, read_file,
()))

diff = datetime.datetime.now() - dt0
print(f"{diff.total_seconds()}")
```

3. Добавьте динамическую точку трассировки для наших системных распределений в "musl`:

```
perf probe -x /usr/lib/ld-musl-x86_64.so.1 'default_malloc n:u32'
perf probe -x /usr/lib/ld-musl-x86_64.so.1 'realloc n:u32'
```

4. Запустите memory.py:

```
perf record -z -e probe_ld:default_malloc -e probe_ld:realloc python
memory.py
```

После завершения работы приложения вы можете проанализировать его выходные данные.

- Обзор созданных задач:

```
perf report --tasks
#      pid      tid     ppid  comm
         0        0       -1 |swapper
      3565     3565       -1 |python
      3565     3567     3565 | python
      3565     3568     3567 |  python
      3565     3569     3567 |  python
      3565     3572     3567 |  python
      3565     3573     3567 |  python
      3565     3574     3567 |  python
      3565     3570     3565 | python
      3565     3571     3570 |  python
      3565     3575     3570 |  python
      3565     3576     3570 |  python
      3565     3577     3570 |  python
      3565     3578     3570 |  python
```

- Различная статистика:

```
perf report --stats
Aggregated stats:
           TOTAL events:      39845
...
probe_ld:default_malloc stats:
          SAMPLE events:      26821
probe_ld:realloc stats:
          SAMPLE events:      12832
```

- Накладные расходы и размер malloc:

```
perf report --stdio -n
```

```
# Samples: 26K of event 'probe_ld:default_malloc'
# Event count (approx.): 26929
#
# Overhead       Samples  Trace output
# ........  ............  ............................
#
    47.05%         12670  (7fddb81be420) n_u32=8225
     1.67%           450  (7fddb81be420) n_u32=4140
     1.41%           380  (7fddb81be420) n_u32=4127
     1.37%           370  (7fddb81be420) n_u32=4116
     ...
```

В нашем примере размер составляет 8225 байт. Выделение памяти происходит в результате чтения файла. Взгляните на объем ресурсов для одного из наших потоков для чтения (у нас было 10 таких потоков):

```
perf report -F pid,overhead,sample,trace --stdio -n
     ...
     3578:python      4.72%          1266  (7fb52688a420) n_u32=8225
     3578:python      0.15%            40  (7fb52688a420) n_u32=4140
     3578:python      0.14%            37  (7fb52688a420) n_u32=4116
     ...
```

Из приведенного выше примера мы видим, что даже при запуске с включенным pymalloc наше простое приложение вызывает malloc() примерно 26 тысяч раз и realloc() 13 тысяч раз во время выполнения, а не только при инициализации. Здесь можно поэкспериментировать с различными функциями выделения памяти. Попробуйте каждый из них следующим образом:

```
for i in dummy rpmallocwrap.so jemalloc.so.2 mimalloc.so.1 mimalloc-
secure.so.1; do
    echo "$i:"
    LD_PRELOAD=/lib/lib$i python memory.py
done
```

В таблице ниже показано, что замена malloc musl даже на mimalloc-secure повышает производительность для этой рабочей нагрузки примерно на 30%.

	musl	glibc	mimalloc	mi-secure	jemalloc	rpmalloc
musl	0.685	-	0.451	0.472	0.438	0.44
glibc	-	0.439	0.473	0.485	0.447	0.449

Функции с повышенной безопасностью

Если безопасность важнее производительности, лучше придерживаться реализации malloc по умолчанию или использовать внешний `mimalloc-secure`. "musl" уже имеет некоторые меры безопасности, которые предотвращают использование ошибок в вызывающем приложении.

Функции безопасности "musl":

- Метаданные выделенной памяти защищены специальными guard страницами.
- Обнаруживает и перехватывает любую попытку освободить слот, который уже свободен, или адрес, который не является частью распределения, полученного malloc.
- Обнаруживает записи после освобождения при следующем вызове в случае, если метаданные становятся непоследовательными.
- Обнаруживает и перехватывает однобайтовые переполнения с произвольными ненулевыми значениями во время перераспределения (realloc) или освобождения (free).

Функции безопасности `mimalloc-secure`:

- Все внутренние страницы mimalloc окружены защитными страницами, а метаданные выделенной памяти также находятся за защитной страницей, поэтому эксплойт переполнения буфера не может достичь метаданных.
- Все указатели свободных списков закодированы с постраничными ключами, которые используются для предотвращения перезаписи известным указателем и для обнаружения повреждения выделенной памяти.
- Двойные свободные места обнаруживаются (и игнорируются).
- Свободные списки инициализируются в случайном порядке, и функция случайным образом выбирает между расширением и повторным использованием на странице, чтобы защититься от атак, основанных на предсказуемом порядке распределения. Аналогично, более крупные

блоки выделенные mimalloc из ОС, имеют рандомизированные адреса.

Axiom Linux
Руководство по реализациям malloc