

Лабораторная работа

ИССЛЕДОВАНИЕ МЕХАНИЗМОВ ВЫДЕЛЕНИЯ ПАМЯТИ

Цели работы:

- изучение системных функций выделения памяти в ОС GNU/LINUX
- получение практических навыков работы с динамической памятью в ОС GNU/LINUX

Методические рекомендации

Управление памятью в Linux

В современных системах процессы не работают с физическими адресами напрямую, у каждого процесса есть свое виртуальное адресное пространство, что позволяет программам при выполнении «предполагать» доступной всю физическую память.

Фактическим отображением виртуальных адресов на физические адреса занимается модуль управления памятью (MMU, Memory Management Unit) - аппаратное устройство, которое управляет памятью и отвечает за трансляцию виртуальных адресов в физические.

Несмотря на то, что наименьшей единицей памяти, которую может адресовать процессор, является машинное слово, ядро рассматривает страницы физической памяти как основные единицы управления памятью. С точки зрения виртуальной памяти, страница - это наименьшая значащая единица.

Размер страницы зависит от архитектуры процессора, обычно он равен 4 Кбайт. Linux также поддерживает возможность Hugepages, позволяющую задавать большие размеры страниц (например, 2 Мбайт).

Ядро сопоставляет каждой странице физической памяти в системе структуру `struct page`. Эта структура определена в файле `<linux/mm.h>` следующим образом:

```
struct page {
    page_flags_t flags;
    atomic_t _count;
    atomic_t _mapcount;
    unsigned long private;
    struct address_space *mapping;
    pgoff_t index;
    struct list_head lru;
    void *virtual;
};
```

Наиболее важные поля этой структуры.

- Поле ***flags*** содержит состояние страницы. Это поле включает следующую информацию: является ли страница измененной (*dirty*) или заблокированной (*locked*) в памяти. Значение каждого флага представлено одним битом, поэтому всего может быть до 32 разных флагов. Значения флагов определены в файле `<linux/page-flags.h>`.

- Поле **_count** содержит счетчик использования страницы. Когда его значение равно нулю, это значит, что никто не использует страницу, и она становится доступной для использования при новом выделении памяти. Код ядра не должен явно проверять значение этого поля, вместо этого необходимо использовать функцию `page_count()`, которая принимает указатель на структуру `page` в качестве единственного параметра. Хотя в случае незанятой страницы памяти значение счетчика `_count` может быть отрицательным (во внутреннем представлении), функция `page_count()` возвращает значение нуль для незанятой страницы памяти и положительное значение - для страницы, которая в данный момент используется. Страница может использоваться страничным кэшем (в таком случае поле `mapping` указывает на объект типа `address_space`, который связан с данной страницей памяти), может использоваться в качестве частных данных (на которые в таком случае указывает поле `private`) или отображаться в таблицу страниц процесса.

- Поле **virtual** - это виртуальный адрес страницы. Обычно это просто адрес данной страницы в виртуальной памяти ядра. Некоторая часть памяти (называемая областью верхней памяти, `high memory`) не отображается в адресное пространство ядра. В этом случае значение данного поля равно `NULL` и страница при необходимости должна отображаться динамически. Необходимо отметить то, что структура `page` связана со страницами физической, а не виртуальной памяти. Поэтому то, чему соответствует экземпляр этой структуры, очень быстро изменяется. Даже если данные, которые содержались в физической странице, продолжают существовать, то это не значит, что они будут всегда соответствовать одной и той же физической странице памяти и соответственно одной и той же структуре `page`, например, из-за вытеснения страниц (`swapping`) или по другим причинам. Ядро использует эту структуру данных для описания всего того, что содержится в данный момент в странице физической памяти, соответствующей данной структуре. Назначение этой структуры – описывать область физической памяти, а не данных, которые в ней содержатся.

Ядро использует структуру `struct page` для отслеживания всех страниц физической памяти в системе, так как ему необходима информация о том, свободна ли страница (т.е. соответствующая область физической памяти никому не выделена). Если страница не свободна, то ядро должно иметь информацию о том, чему принадлежит эта страница. Возможные обладатели: процесс пространства пользователя, данные в динамически выделенной памяти в пространстве ядра, статический код ядра, страничный кеш (`page cache`) и т.д.

Адресация виртуальной памяти

Linux использует трехуровневую структуру таблицы страниц, состоящую из следующих типов таблиц (каждая отдельная таблица имеет размер, равный одной странице):

- Каталог страниц. Активный процесс имеет каталог страниц, размер которого равен одной странице. Каждая запись в каталоге страниц указывает на одну страницу промежуточного каталога страниц. Каталог страниц активного процесса должен находиться в активной памяти.

- Промежуточный каталог страниц. Промежуточный каталог страниц может охватывать несколько страниц. Каждая запись промежуточного каталога указывает на одну страницу таблицы страниц.
- Таблица страниц. Таблица страниц также может охватывать несколько страниц. Каждая запись таблицы страниц указывает на одну виртуальную страницу процесса.

При использовании трехуровневой структуры таблицы страниц виртуальный адрес рассматривается как состоящий из четырех частей (рисунок 1). Левое поле используется в качестве индекса в каталоге страниц; следующее поле служит в качестве индекса в промежуточном каталоге страниц. Третье поле представляет собой индекс таблицы страниц, а четвертое - смещение в пределах выбранной страницы памяти.

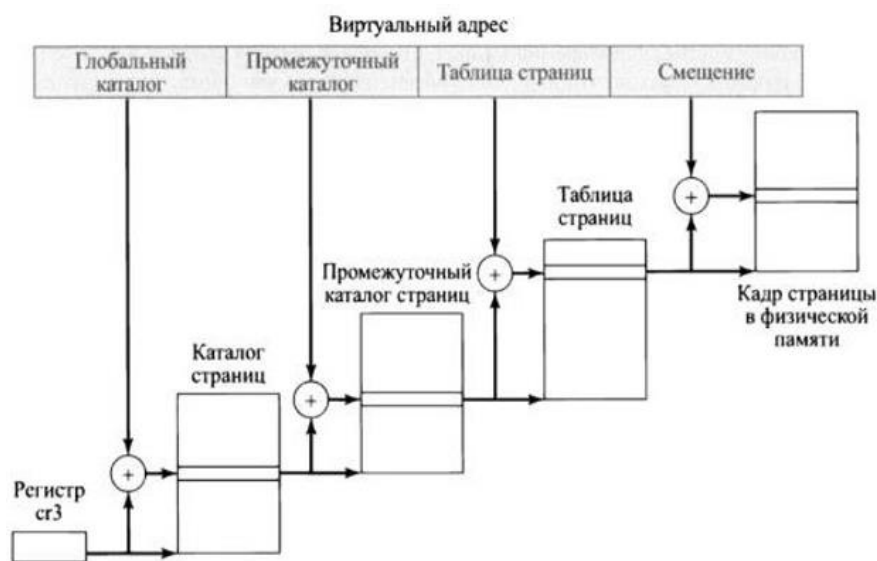


Рисунок 1. Трансляция адреса в схеме виртуальной памяти Linux

Структура таблицы страниц Linux платформонезависима и разработана для работы с 64-разрядным процессором Alpha, который обеспечивает аппаратную поддержку трех уровневой страничной организации. При использовании 64-разрядных адресов использование только двух уровней может привести к тому, что таблицы и каталоги страниц будут очень большими. 32-разрядная архитектура x86 обладает только двухуровневым механизмом страничной организации, и программное обеспечение Linux использует двухуровневую схему путем определения размера промежуточного каталога, равного одной странице. Все ссылки на дополнительный уровень косвенности устраняются оптимизатором во время компиляции, а не во время выполнения. Таким образом, при использовании обобщенного трехуровневого дизайна на платформах, которые аппаратно поддерживают только два уровня, снижение производительности не наблюдается.

Выделение и освобождение динамической памяти в процессе

При создании, например, статического массива фиксированного размера под него выделяется определенная память, размер которой известен на этапе компиляции. Например, создание массива с пятью элементами в исходном тексте программы:

```
double numbers[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
```

Для массива *numbers* выделяется память $5 * 8$ (размер типа `double`) = 40 байт так как заранее точно известно, сколько в массиве элементов и сколько он занимает памяти.

В части случаев бывает необходимо, чтобы количество элементов и соответственно размер выделяемой памяти для массива определялись динамически в зависимости от некоторых условий (например, при вводе пользовательских данных). В этом случае для создания массива используется динамическое выделение памяти.

Для управления динамическим выделением памяти используется ряд функций, которые определены в заголовочном файле `stdlib.h`:

`void *malloc(unsigned s);`

выделяет память длиной в *s* байт и возвращает указатель на начало выделенной памяти. В случае неудачного выполнения возвращает `NULL`

`void *calloc(unsigned n, unsigned m);`

выделяет память для *n* элементов по *m* байт каждый и возвращает указатель на начало выделенной памяти. В случае неудачного выполнения возвращает `NULL`

`void *realloc(void *bl, unsigned ns);`

изменяет размер ранее выделенного блока памяти, на начало которого указывает указатель *bl*, до размера в *ns* байт. Если указатель *bl* имеет значение `NULL`, то есть память не выделялась, то действие функции аналогично действию `malloc`

`void *free(void *bl);`

освобождает ранее выделенный блок памяти, на начало которого указывает указатель *bl*.

Рассмотрим пример применения функции `malloc()`:

```
#include <stdio.h>  
#include <stdlib.h>           // для подключения функции malloc  
int main(void)  
{  
    int *ptr = malloc(sizeof(int)); // выделяем память для переменной  
    int if(ptr != NULL)  
    {  
        *ptr = 42;           // помещаем значение в выделенную память  
        printf("%d \n", *ptr); // выводим результат размещения значения  
    }  
    free(ptr);               // освобождаем память  
}
```

Чтобы узнать, сколько байтов надо выделить, передаем в функцию **malloc()** размер типа **int**, полученный с помощью оператора **sizeof**, в результате успешной работы получаем указатель **ptr**, который указывает на выделенную память

Переменная типа **int** на большинстве архитектур занимает 4 байта, и в большинстве случаев будет выделяться память объемом в 4 байта.

Теоретически мы можем столкнуться с тем, что функции **malloc()** не удастся выделить требуемую память, и тогда она возвратит **NULL**. Чтобы избежать подобной ситуации перед использованием указателя мы можем проверять его на значение **NULL**:

Хотя память была освобождена с помощью функции **free()**, указатель сохраняет свой адрес, и теоретически возможно обратиться к памяти по данному указателю. Однако полученные значения уже будут неопределенными и недетеминированными, подобное обращение может привести к ошибке. Поэтому после освобождения памяти можно также установить значение **NULL** для указателя:

ptr = NULL;

Задания на лабораторную работу

***Обратите внимание!** В работе предполагается использование изученных ранее системных средств: сигналов, служб. Для выполнения п.4 с целью наглядной демонстрации результатов допустимо использование python для построения графиков.*

Задание 1. Напишите программу на C/C++, работающую в одном из 2-х режимов, режим работы задается при запуске в качестве аргумента командной строки:

- Режим 1 (P1). Сбалансированное выделение/освобождение памяти
- Режим 2 (P2). Контролируемая "утечка" памяти

В P1 выделяйте блоки **malloc()** и инициализируйте их, при этом программа после выделения нового блока должна освобождать предыдущий. В P2 выполняйте освобождение не всех выделенных блоков, а, например, каждого 5го. Выделение и освобождение памяти в P1 и P2 выполняйте параллельно с задержкой 100 – 1000 нс в бесконечном цикле.

Предусмотрите прерывание процесса по сигналу, направленному асинхронно, для освобождения всей динамически выделенной памяти.

Задание 2. Напишите **bash**-скрипт для автоматизации сбора информации о виртуальной памяти целевого процесса (предусмотреть передачу **pid** процесса).

Скрипт должен с использованием команды **rmap** получать карту виртуальной памяти целевого процесса (при желании можно также воспользоваться информацией из **/proc/** для целевого процесса). Полученную карту необходимо сохранять в отдельную директорию для карт памяти, например, **/var/log/my_mem_maps/**.

Имя файла должно быть задано по шаблону **map_<pid>_<date&time>[_<mode>]**, где **<pid>** - идентификатор процесса, переданный в программу и соответствующий процессу, для которого создается карта, **<date&time>** - дата и время создания карты в формате **уууу-мм-дд_h:m:s**, **<mode>** - режим работы программы P1 или P2.

Задание 3. Автозапуск скрипта по таймеру.

Создайте службу и таймер, запускающие каждые 30 секунд на выполнение скрипт из задания 2, проверьте статус службы и таймера. Получите 10-15 файлов с картами памяти для программы из п.1. для каждого из режимов работы P1 и P2.

Включите в отчет результаты и скриншоты по всем заданиям, выводы, ответы на контрольные вопросы.

Дополнительное задание

Анализ карт памяти.

Проанализируйте полученные результаты, с помощью самостоятельно написанной программы на языке Python, строящей графики по данным карт виртуальной памяти из задания 3 (для N последних файлов карт в указанной при запуске директории при выбранном режиме работы, N и путь до директории передаются в скрипт при запуске).

Выявите изменение размера кучи, при необходимости проведите корректировку размеров выделяемой `malloc()` памяти.

Для анализа извлекайте, например, следующие данные:

- временную метку из имени файла
- размер кучи и стека из вывода `rtop`
- общие метрики памяти (VSZ, RSS)

Рассчитайте начальные, конечные, максимальные и минимальные значения полученных данных, средние значения и отклонение СКО.

Постройте график изменения кучи и RSS по времени.

Включите в отчет результаты и скриншоты, полученные графики и выводы.

Контрольные вопросы

1. Что такое страница памяти и какие размеры страниц поддерживаются в Linux?
2. Какие существуют типы выделения памяти в программах на C?
3. Что такое виртуальная память процесса и какие основные сегменты она включает (stack, heap, data, text)?
4. Как функции `malloc()` и `free()` взаимодействуют с ядром ОС?
5. Всегда ли вызов `malloc()` приводит к системному вызову?
6. Что показывает карта виртуальной памяти процесса и как идентифицировать в ней кучу (heap)?
7. Почему размер кучи в картах памяти может увеличиваться, но не уменьшаться даже после вызовов `free()`?
8. Какие метрики памяти наиболее важны для анализа при исследовании работы программы с динамической памятью?
9. Как отличить штатный рост потребления памяти от утечки памяти при анализе серии карт памяти?
10. Почему Resident Set Size (RSS) может отличаться от размера кучи в картах памяти?