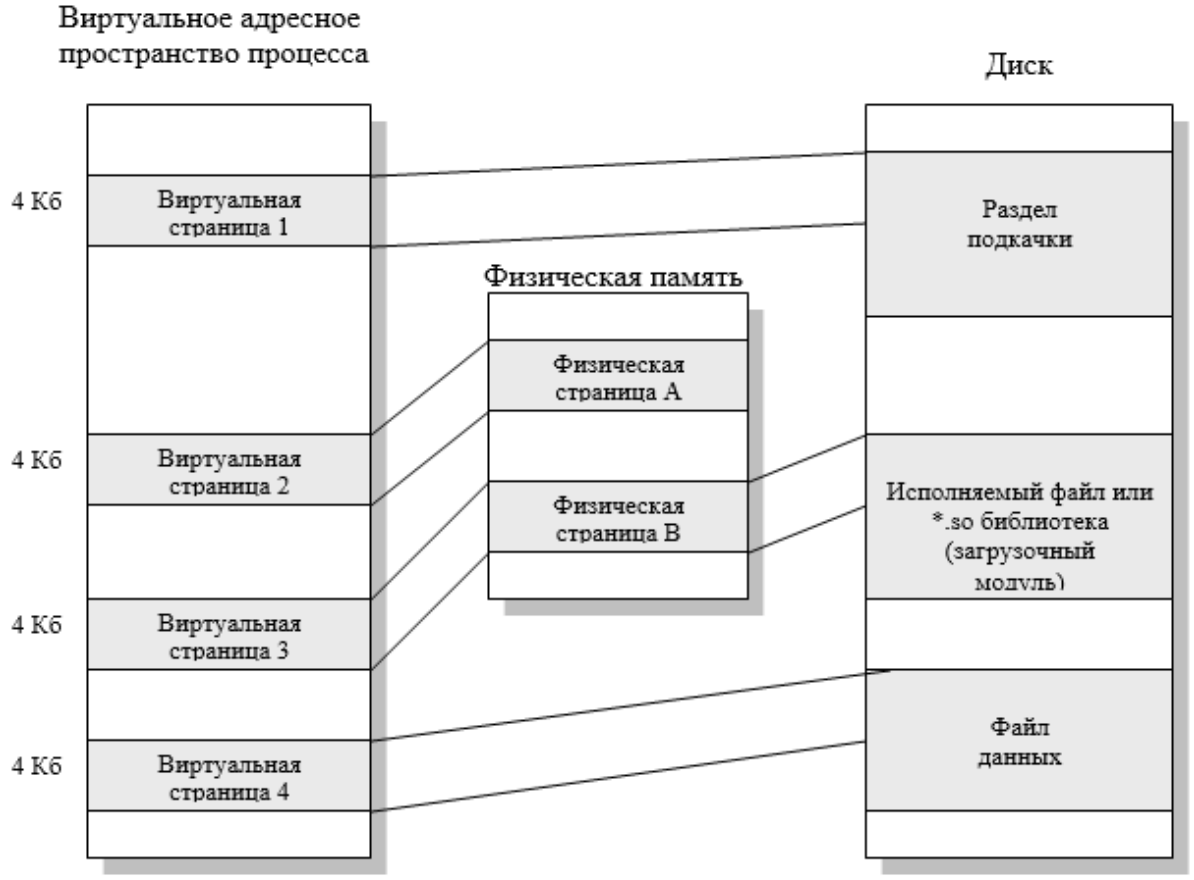
Лабораторная работа №4. Виртуальная память

Цель работы: получение практических навыков по использованию механизмов работы с виртуальной памятью ОС Linux.

Теоретические сведения

Типы памяти

На рисунке ниже представлена взаимосвязь виртуального адресного пространства процесса с физической и внешней памятью.



Физическая память (physical memory) - это реальные микросхемы RAM, установленные в компьютере. Каждый байт физической памяти имеет физический адрес (physical address), который представляет собой число от нуля до числа на единицу меньшего, чем количество байтов физической памяти. Например, ПК с установлен¬ными 64 Мб RAM, имеет физические адреса 0x0000000-0x4000000 в шестнадцатеричной системе счисления, что в десятичной системе будет 0-67108863. Физическая память (в отличие от раздела подкачки и виртуальной памяти) является исполняемой (executable), то есть памятью, из которой можно читать и в которую центральный процессор может посредством системы команд записывать данные.

Виртуальная память (virtual memory) - это просто набор чисел, о которых говорят как о виртуальных адресах. Программист может использовать виртуальные адреса, но программа не способна по этим адресам непосредственно обращаться к данным, поскольку такой адрес не

является адресом реального физического запоминающего устройства, как в случае физических адресов и адресов файла подкачки. Для того чтобы код с виртуальными адресами можно было выполнить, такие адреса должны быть отображены на физические адреса, по которым действительно могут храниться коды и данные. Эту операцию со стороны процессора выполняем специальный модуль – *MMU (Memory management unit)*, поддержка поторого включена в *менеджер памяти* со стороны ОС Linux.

Как известно, наименьший адресуемый блок памяти - байт. Однако самым маленьким блоком памяти, которым оперирует менеджер памяти ОС Linux, является **страница (page) памяти**, называемая также страничным блоком (page frame) памяти. На компьютерах с процессорами Intel и 32-х разрядной ОС размер страницы памяти равен 4 Кб.

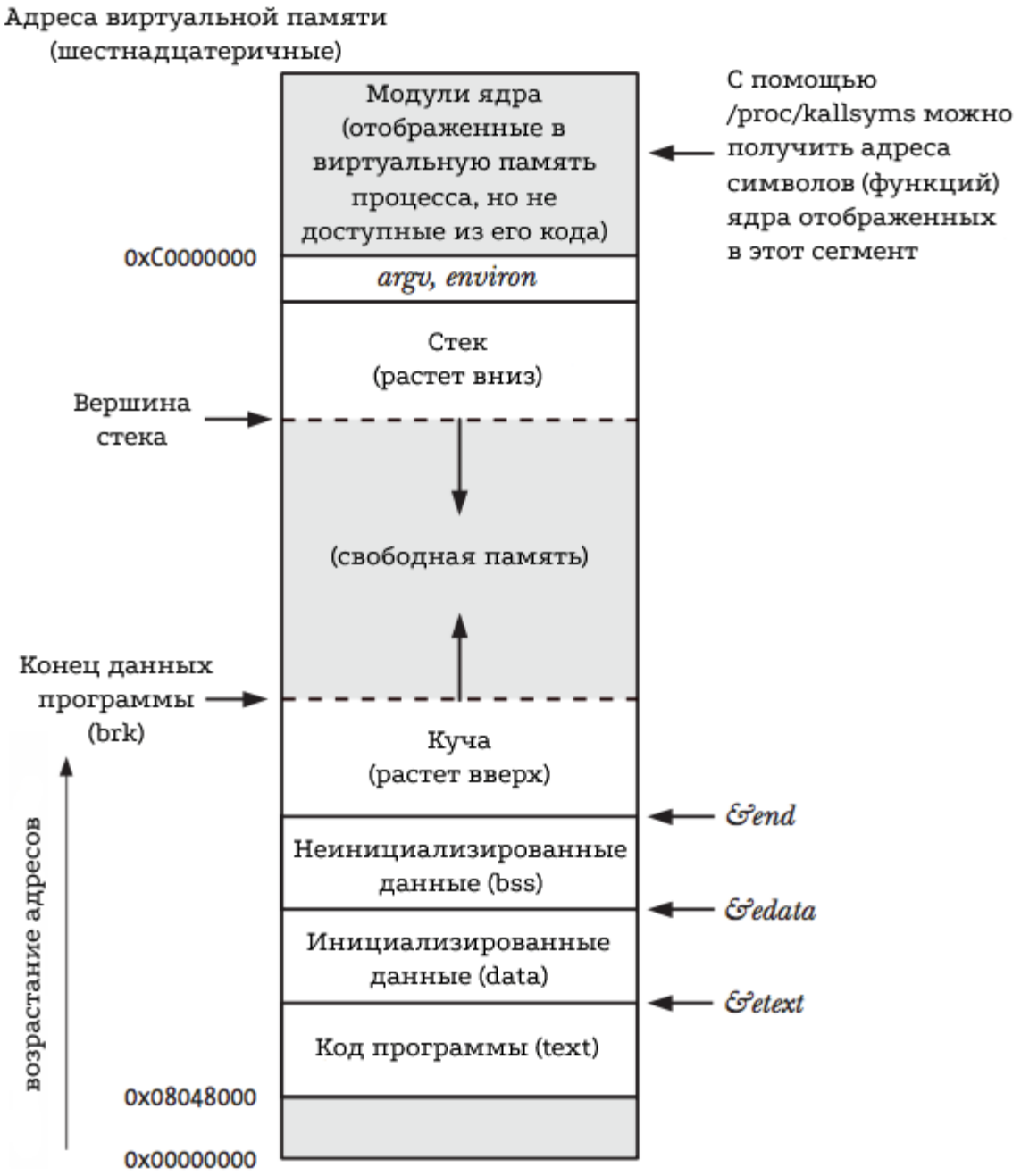
Раздел подкачки (swap partition) находится на жестком диске и используется для хранения данных и программ точно так же, как и физическая память, но его объем обычно выбирается превышающим объем физической памяти. ОС Linux использует раздел подкачки для хранения информации, которая не помещается в RAM, производя, если нужно, обмен страниц между разделом подкачки и RAM.

Карта адресного пространства процесса

Ядро упорядочивает страницы в блоки (сегменты), которые имеют общие определенные свойства (например, права доступа). В каждом процессе можно найти следующие сегменты памяти:

- Сегмент кода (.text) - содержит программный код процесса, строковые литералы, постоянные переменные и другие данные, доступные только для чтения. В Linux этот сегмент помечен как доступный только для чтения и отображается непосредственно из объектного файла (исполняемого файла программы или библиотеки).
- Сегмент данных (.data) - содержит инициализированные глобальные и статические переменные, которые могут быть изменены в процессе выполнения программы.
- Сегмент неинициализированных данных (.bss) - содержит неинициализированные глобальные переменные. Эти переменные инициализируются нулевыми значениями, в соответствии со стандартом С.
- Сегмент данных, или куча, содержит динамическую память процесса. Этот сегмент доступен для записи и может увеличиваться или уменьшаться в размерах. malloc() может удовлетворять запросы памяти из этого сегмента.
- Стек - содержит стек выполнения процесса, который динамически увеличивается и уменьшается по мере увеличения и уменьшения глубины стека. Стек выполнения содержит локальные переменные и данные, возвращаемые функцией. В многопоточном процессе на каждый поток приходится один стек.

На рисунке ниже приведено адресное пространство процесса в ОС Linux.



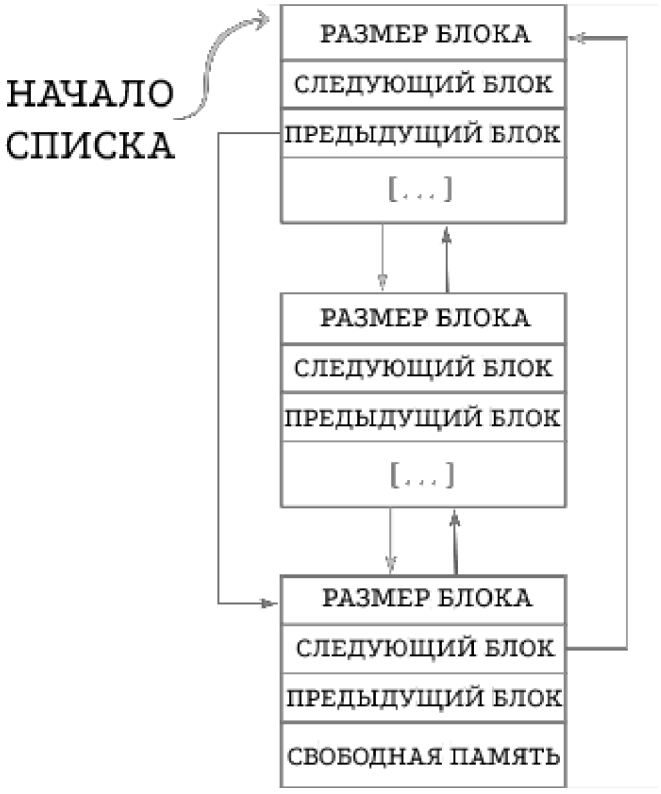
Реализация менеджера кучи

Поскольку все элементарные подходы к выделению дополнительной памяти процессам имеют явные недостатки, ограничивающие среды и задачи, для решения которых эти подходы могут применяться, были выработаны различные методики построения менеджеров кучи, на основе эвристик и подходящие для использования в широком диапазоне задач. В данном разделе приводится упрощенное описание одного из таких механизмов

выделения памяти на куче, реализованный в стандартной библиотеке языка С.

При запросе памяти программой возвращается, выровненный на 8 байт в случае с 32-разрядной или 16 байт в случае с 64-разрядной ОС, адрес памяти выделенной на куче. Данное выравнивание не имеет большого смысла, если в выделенном участке памяти будет храниться строка символов или массив байтов. Однако смысл в выравнивании появляется при размещении по выделенному адресу структур составленных из элементарных типов большей разрядности, поскольку это позволит делать процессору один запрос к памяти вместо двух, что сэкономит порядка 200 циклов этого процессора.

Для обеспечения эффективности работы менеджера кучи выделенные участки памяти организуются в двусвязный список блоков.



Каждый блок включает выделенный участок памяти и служебные данные, которые предваряют этот участок памяти. Служебные данные содержат размер выделенного блока памяти, битовые флаги, и указатели на предыдущий и следующий блок в списке ранее выделенных блоков памяти. Поскольку размер выделяемого участка памяти округляется вверх на то же значение, что и выравнивание, младшие биты данного поля структуры блока могут быть использованы для хранения флагов. Из трёх флагов использующихся в оригинальной реализации менеджера кучи здесь ограничимся единственным флагом выделения памяти вне кучи.

```
typedef struct chunk {
  union {
    unsigned long size;
    unsigned long flags;
  };
  struct chunk *next, *prev;
  struct {} __attribute__((aligned(16))) __padding;
  char data[0];
} chunk_info;

#define ALIGN_DOWN(base, align) ((base) & -((__typeof__(base)) (align)))

#define CHUNK_M 0x02 // Mmap'd chunk
#define get_chunk_size(chunk) ((chunk)->size & ~0x7UL)
#define set_chunk_size(chunk, sz) ((chunk)->size = ALIGN_DOWN(sz, 8) |
get_chunk_flags(chunk))
#define check_chunk_flags(chunk, f) ((chunk)->flags & f)
#define get_chunk_flags(chunk) ((chunk)->flags & 0x7UL)
#define set_chunk_flags(chunk, f) ((chunk)->flags = get_chunk_size(chunk)
| f)
```

Алгоритм выделения памяти менеджером кучи состоит из следующих шагов:

1. Осуществляется поиск подходящего участка памяти среди освобожденных блоков. Для этого удобно поддерживать упорядоченность в списке блоков по их размеру. Связный список блоков пополняется при каждом вызове функции *free*.
2. Если подходящего блока не нашлось, то менеджер памяти создаёт блок из ещё не выделенной памяти в конце кучи. Это происходит также и при первой попытке выделения памяти в программе.
3. В случае, если вся память в куче уже занята блоками, то производится увеличение размера кучи с помощью системного вызова *brk* или *sbrk*, которые также можно использовать для получения адреса начала кучи. Системный вызов *sbrk* возвращает этот адрес при его первом вызове, а *brk* - при нулевом значении аргумента. Увеличение сегмента кучи осуществляется за счёт включения дополнительных страниц памяти в процесс.
4. В случае, если уже использован весь диапазон адресов (дальнейшее расширение кучи упирается в отображения в память, разделяемые библиотеки или стек), доступных для выделения под кучу, выделение памяти завершается ошибкой.

Если запрошенный размер участка памяти превышает 32 мегабайта, то блок памяти выделяется с помощью системного вызова *mmap*, а в самом блоке памяти устанавливается флаг, что выделение производилось вне кучи и выделенная память должна быть освобождена с помощью системного вызова *munmap*. При выделении пмяти таким образом в *mmap* необходимо передать комбинацию флагов MAP_PRIVATE и MAP_ANONYMOUS.

Задания

1. Реализовать менеджер кучи по приведенному в теоретической части описанию. Обеспечить к нему интерфейс в виде двух функций аналогичных *malloc* и *free*. Проверить работоспособность работы менеджера кучи последовательно выделяя участки памяти различного размера от нескольких десятков байт до нескольких гигабайт.

   Замечание 1: при разработке и тестировании менеджера кучи следует исключить использование функции *mallloc* предоставляемой стандартной библиотекой языка С.

2. По выполненной работе нужно оформить отчет, содержащий описание ключевых аспектов реализации и демонстрацию работы написанных программ.

Ссылки

1. Arm Heap Exploitation Part 1: Understanding the GLIBC heap implementation: https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/
2. Arm Heap Exploitation Part 2: Understanding the GLIBC heap implementation: https://azeria-labs.com/heap-exploitation-part-2-glibc-heap-free-bins/
3. Memory Addressing: https://notes.shichao.io/utlk/ch2/
4. A printf / sprintf Implementation for Embedded Systems: https://github.com/mpaland/printf