ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

КАФЕДРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

ЛАБОРАТОРНАЯ РАБОТА № 2
«ДИНАМИЧЕСКОЕ УПРАВЛЕНИЕ ПАМЯТЬЮ В ПРИЛОЖЕНИЯХ, НАПИСАННЫХ НА ЯЗЫКЕ СИ»

Автор: С.Н. Мамойленко

Новосисбирск - 2013

Оглавление

Цель работы

Изучить принципы динамического управления оперативной памятью в процессе выполнения программ, написанных на языке Си под управлением операционной системы GNU/Linux.

Теоретическое введение

1. Общие сведения о динамическом управлении памятью

Для выполнения любого программного обеспечения на ЭВМ или вычислительной системе необходимо наличие как минимум двух инструментов: исполняющего (обычно это вычислительное ядро, процессор, группа процессоров и т.п.) и хранящего информацию (оперативная память, внешние накопители информации и т.п.). Очевидно, что управлением всеми устройствами и ресурсами ЭВМ занимается системное программное обеспечение (даже в случае выполнения всего одной программы, монопольно владеющей всеми ресурсами ЭВМ). Управление ресурсами – это сложный процесс, требующий решения комплекса взаимосвязанных задач. Одной из таких задач является управление оперативной памятью.

В зависимости от архитектуры ЭВМ или вычислительной системы и режима их функционирования управление оперативной памятью может реализовываться на разных уровнях: выделение памяти процессам, реализация виртуальных адресных пространств процессов, управление памятью внутри адресных пространств процессов и т.п. Рассмотрим задачу управления памятью внутри адресных пространств процессов подробнее.

Для хранения информации в программном обеспечении используются различные структуры данных: простые (скалярные) величины, массивы, объединения, структуры, указатели и т.п. Выделение памяти для хранения структур данных основывается на двух взаимодополняющих стратегиях. Первая стратегия («***статическое выделение памяти***») предполагает, что все необходимые структуры данных заранее определяются на этапе разработки программного обеспечения и в процессе его выполнения не изменяются, а меняется лишь только информация, хранимая в этих структурах. Вторая стратегия предполагает, что обрабатываемая информация заранее не определена и структуры данных, необходимые для её хранения, ***динамически*** создаются в процессе выполнения программного обеспечения.

Статическое распределение памяти задается в исходных кодах программного обеспечения и формируется компилятором в процессе подготовки исполняемого файла. Выделение оперативной памяти в этом случае осуществляется один раз при загрузке исполняемого файла.

Динамическое распределение памяти предполагает, что программному обеспечению предоставляется некоторый объем изначальное нераспределённой оперативной памяти и средства управления этой памятью. Очевидно, что в этом случае системное программное обеспечение отвечает за выделение памяти в адресном пространстве процесса, её освобождение, перераспределение и хранение информации о текущем её состоянии. Память, используемая для динамического выделения и структура данных, используемая для хранения информации о выделении памяти, называется ***кучей***[1]. Другими словами, куча — это длинный отрезок адресов памяти, поделенный на подряд идущие блоки различных размеров [wiki].

2. Функции динамического управления памятью библиотеки GLIBC.

В GNU/Linux широко используется библиотека функций GLIBC управление динамической памятью в которой реализуется с помощью функций: `malloc`, `free`, `realloc`, `calloc` и т.д. (см. рисунок 1). Пример использования функций показан ниже (см. рисунок 2).

[1] Следует отметить, что значение этого термина отличается от структуры хранения данных «куча», которая определяет очередь с приоритетами.

Первая версия функций динамического управления памятью была разработана Doug Lea профессором университета Oswego (New York) и называлась она dlmalloc. В текущей версии библиотеки GLIBC используется версия, разработанная Wolfram Gloger, которая основывается на dlmalloc и называется ptmalloc[2]. Основные отличия заключаются в поддержке динамического выделения памяти для многопоточных приложений.

Функция `malloc` выделяет участок памяти заданного размера и возвращает указатель на его начало. Размер выделяемого блока указывается в байтах. В случае ошибки (неверно указан размер или выделить память заданного размера невозможно) функция возвращает `NULL`. Содержимое выделенного участка памяти не определено.

```
#include <malloc.h>
#include <stdlib.h>

void *malloc (size_t size);
void *calloc (size_t nmemb, size_t size);
void free (void *ptr);
void *realloc (void *ptr, size_t size);


void malloc_stats (void);
struct mallinfo mallinfo(void);

void malloc_trim(size_t pad);
int mallopt(int param, int value);

void (* volatile __malloc_initialize_hook)(void);
void *(*__malloc_hook)(size_t size, const void *caller);
void *(*__realloc_hook)(void *ptr, size_t size, const void *caller);
void (*__free_hook)(void *ptr, const void *caller);
```

Рисунок **1** – Некоторые функции динамического управления памятью библиотеки GLIBC

```
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main (void){
  char * str;
  int * array;
  str = (char *) malloc (15);
  array = (int *) calloc (sizeof(int), 10);
  if (str == NULL) { printf ("Error!\n"); exit(1); }
  strcpy (str, "Первый раз");
  array[4] = strlen(str);
  printf ("Строка [%s] имеет длинну %d символов\n", str, array[4]);
  free (str); free (array);
  return (0);
}
```

Рисунок **2** – Пример использования функций динамического управления памятью

[2] Подробнее о библиотеке ptmalloc можно прочитать на официальном сайте - http://www.malloc.de/en/.

Если необходимо выделить участок памяти, достаточный для хранения последовательности из нескольких одинаковых структур данных (массива), то следует использовать функцию `calloc`. Результат работы функции `calloc` аналогичен результату работы функции `malloc`, за исключением того, что выделенная память изначально заполняется нулями.

Изменение размера выделенного участка памяти производится с помощью функции `realloc`. При этом, если новый размер меньше ранее выделенного, то данные обрезаются до указанного размера. Если требуется увеличить ранее выделенный блок, то данные копируются во вновь выделенный блок. В результате функция возвращает указатель на новый блок или `NULL` в случае ошибки.

Функция `free` освобождает участок памяти, который ранее был выделен функциями `malloc`, `calloc`, `realloc`. В случае возникновения ошибки функция принудительно завершает процесс.

Дополнительно к этим функция в библиотеке GLIBC присутствуют средства настройки, описание которых представлено ниже.

3. Адресное пространство процесса и область динамической памяти

Прежде, чем говорить о динамическом распределении памяти, следует понять каким образом устроено адресное пространство процесса (в условиях заданной архитектуры ЭВМ и её аппаратурных особенностей). Распределение памяти (карту памяти) действующего процесса можно посмотреть, например, с использованием виртуальной файловой системы `/proc` в файле `maps`, располагающемся в каталоге с соответствующим номером PID (см. рисунок **3**[3]).

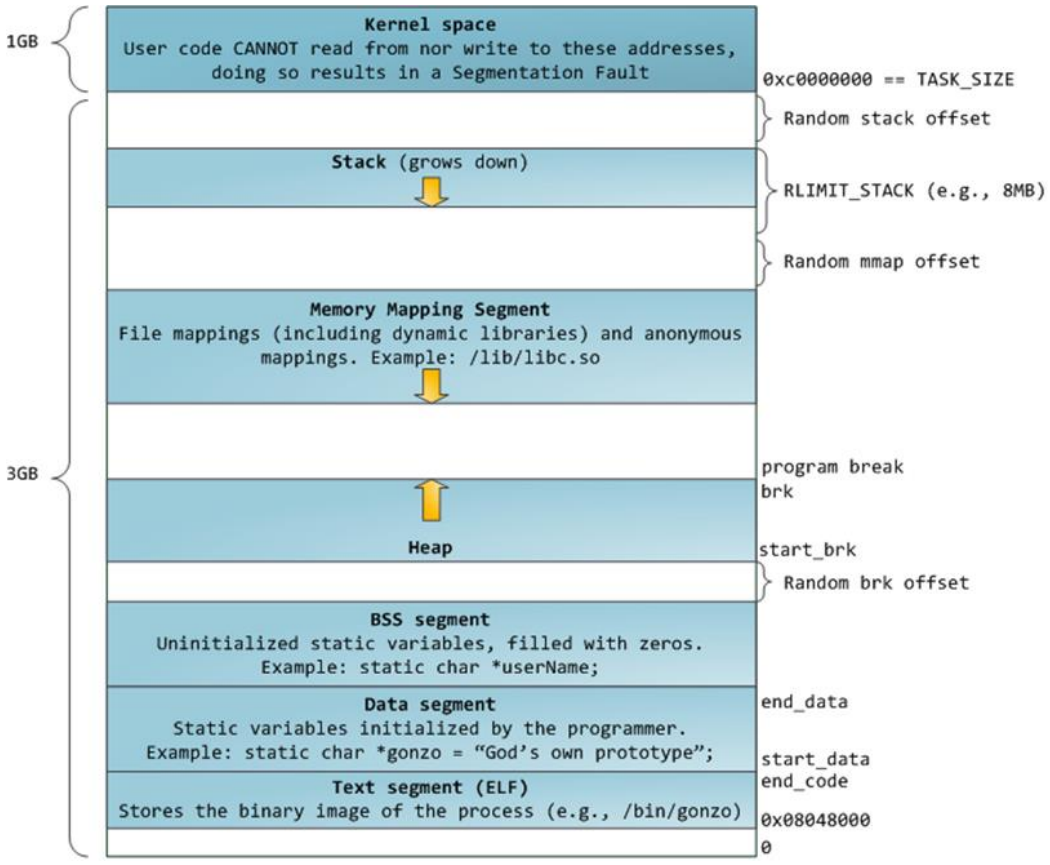

Рисунок **3** – Виртуальное адресное пространства процесса в системе GNU/Linux (в архитектуре Intel IA32).

[3] Рисунок позаимствован из статьи http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory.

В системе GNU/Linux, функционирующей на ЭВМ с архитектурой IA32, все процессы работают в рамках собственного виртуального адресного пространства размером 4 ГБайт. При этом старший 1Гбайт занимает пространство ядра, а остальные 3 ГБайт – пространство пользователя, в котором размещается исполняемая программа (сегмент text), статически распределённая память (сегменты data и BSS), стек (сегмент Stack) и оставшаяся часть - область динамической памяти.

Последнюю область адресного пространства процесса условно можно разделить на две подобласти – `brk/sbrk` (начинается с младших адресов и увеличивается вверх) и область `mmap` (начинается со старших адресов и увеличивается вниз). Подобласть `brk/sbrk` используется только для динамического выделения памяти выполняющейся программе. Подобласть `mmap` предназначена для отображения в памяти файлов (в том числе разделяемых библиотек) и обеспечения связывания адресных пространств процессов (разделяемая память). Отображение файлов в виртуальной памяти процесса по сути является буферизованным чтением/записью конкретного файла с произвольным доступом. С применением специального файла (`/dev/zero`) такой способ доступа к памяти позволяет использовать это адресное пространство как дополнение к подобласти `brk/sbrk`.

Библиотека GLIBC использует область `brk/sbrk` в однопоточных приложениях для выделения блоков небольшого размера. В многопоточных приложениях для блоков небольших размеров в каждой нити в подобласти `mmap` выделяется свое пространство (в библиотеке оно называется `arena`), работа с которым аналогична работе с областью `brk/sbrk`. Память для «больших» блоков выделяется библиотекой в подобласти `mmap`.

В версии 2.18 библиотеки GLIBC[4] минимальный размер блока, начиная с которого он относится к «большим» задается константой `M_MMAP_THRESHOLD`, которая по умолчанию имеет значение, равное 128*1204 байт (128 Кбайт). О том как изменить значение этой константы будет сказано ниже.

4. Структура блоков динамически выделяемой памяти

Динамическая память распределяется в виде блоков - последовательно расположенных ячеек памяти. В терминах библиотеки GLIBC такой блок называется `chunk`.

Блоки памяти выделяются и освобождаются по запросам (вызовам функций `malloc/free`) исполняемой программы. В результате блоки могут находится в одном из двух состояний: «свободен» и «выделен». Для описания блока, его состояния и контроля его целостности используется служебная информация, размещаемая в оперативной памяти перед и после блока (см. рисунок **4**).

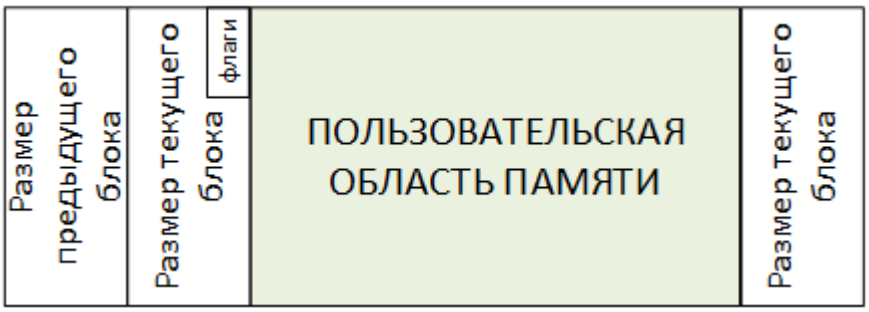

Рисунок **4** – Структура блока динамически выделенной памяти.

В заголовке блока указывается размер предыдущего блока и размер текущего блока. После пользовательского блока также хранится размер текущего блока. Очевидно, что два последовательных блока пересекаются по первому полю заголовка и полю завершения. Пример использования этих полей приведен ниже (см. рисунок **5**).

Размер пользовательской области блока выравнивается по размеру страницы памяти.

[4] Подробнее о библиотеке можно прочитать на официальном сайте http://www.gnu.org/software/libc/libc.html.

```
#include <stdio.h>
#include <malloc.h>

struct malloc_chunk { size_t prev_size, size; };

int main (void){
  char * ptr_user;
  struct malloc_chunk * ptr_chunk;
  ptr_user = malloc (10);
  ptr_chunk = (struct malloc_chunk *)(
                                (char*)(ptr_user) - 2*sizeof(size_t));
  printf ("size = %d\n", (int)(ptr_chunk->size));
  free (ptr_user);
  return (0);
}
```

Рисунок **5** – Определение размера динамически выделенного блока.

5. Конфигурирование подсистемы динамического управления памятью в GLIBC

Поведение функции `malloc` может быть настроено с помощью нескольких переменных (см. таблицу 1). Изменить значения этих переменных можно с помощью функции `mallopt` (см. рисунок **6**). Вывести статистическую информацию о подсистеме памяти можно с помощью функций `malloc_stats` и `mallinfo`. Журнал вызова функций динамического выделения памяти можно получить с помощью функции `mtrace`.

Таблица 1. Переменные, управляющие поведением функций динамического выделения памяти.

Переменная	Назначение	Значение
M_MXFAT	Количество блоков небольшого размера, освобождение которых не приводит освобождению оперативной памяти	64
M_TRIM_THRESHOLD	Объем свободной памяти в блоке brk/sbrk, при котором размер блока сокращается	128*1024
M_MMAP_THRESHOLD	Минимальный размер блока, который выделяется с помощью mmap	128*1024
M_MMAP_MAX	Максимальное количество блоков, которые могут быть выделены с помощью mmap	65536

```
#include <malloc.h>
#include <stdlib.h>

int main (void){
 int i;
 mallopt (M_MMAP_THRESHOLD, 900*1024);
 for (i = 0; i < 1000; i ++) malloc (i*1024);

 malloc_stats();
 return (0);
}
```

Рисунок **6** – Настройка подсистемы выделения динамической памяти.

6. Алгоритмы поиска свободных блоков памяти.

Основная проблема реализации функций динамического управления памяти заключается обеспечении эффективного поиска свободных блоков. Очевидно, что чем дольше происходит выделение памяти, тем больше накладных расходов в процессе исполнения приложения.

В библиотеке GLIBC все свободные блоки помещаются в пакеты согласно их размера. Всего поддерживается 64 пакета с блоками размером от 1 до 8 байт, 32 пакета с блоками от 9 до 64 байт, 16 пакетов с блоками от 65 до 512 байт, 8 пакетов с блоками до 4096 байт, 4 пакета с блоками до 32768 байт, 2 пакета с блоками до 262144 байт и 1 пакет с блоками больше 262144 байт.

Внутри каждого пакета блоки размещены в двунаправленном списке. Для хранения списка используется пользовательское пространство блоков (адреса следующего и предыдущего блока хранятся в пользовательском пространстве блока). Блоки размером до 4096 байт в своих списках никак не сортируются, а остальные блоки отсортированы по убыванию.

Для поиска необходимого блока используется алгоритм «Наилучший подходящий», суть которого заключается в том, что среди имеющихся свободных блоков находится тот, размер которого максимально приближен к требуемому размеру блока (с учетом размеров служебной информации и выравнивания размера блока).

Если найден блок большего размера, чем это необходимо и оставшейся части достаточно для того, чтобы создать свободный блок минимального и большего размера, то найденный блок делится на две части: свободный блок и выделенный блок. Свободный блок вновь помещается в соответствующий пакет, а выделенный блок возвращается пользователю.

7. Перехват вызовов функций динамического управления памятью в библиотеке GLIBC

Пользователю функций динамического распределения памяти в библиотеке GLIBC предоставлена возможность перехвата вызовов этих функций (см. рисунок **7**). Эта возможность обычно используется для отладки приложений и анализа качества их работы с динамической памятью.

```
#include <malloc.h>

static void *(*old_malloc_hook) (size_t, const void *);
static void  (*old_free_hook) (void *, const void *);

static void * my_malloc_hook (size_t size, const void *caller){
  void *result;
  __malloc_hook = old_malloc_hook;
  result = malloc (size);
  old_malloc_hook = __malloc_hook;
  __malloc_hook = my_malloc_hook;
  return result;
}

static void my_free_hook (void * ptr, const void * caller){
  __free_hook = old_free_hook;
  free(ptr);
  old_free_hook = __free_hook;
  __free_hook = my_free_hook;
}

static void my_init_hook (void){
  old malloc hook =   malloc hook;
```

```
  old_free_hook = __free_hook;
  __malloc_hook = my_malloc_hook;
  __free_hook = my_free_hook;
}

void (* volatile __malloc_initialize_hook) (void) = my_init_hook;

int main (void) {
  char *p;
  p = malloc (10);
  free (p);
  return 0;
}
```

Рисунок **7** – Перехват вызовов функций динамического распределения памяти.

Задание на лабораторную работу

1. Напишите программу, которая выделяет 1000 блоков памяти по 120 байт каждый и выводит статистику (malloc_stats). Сколько блоков памяти было выделено в разделе «малых блоков»? Измените программу так, чтобы размер блока рассчитывался как i*1024, где i – номер итерации цикла. Повторите эксперимент и ответьте на тот же вопрос.
2. Напишите программу, которая перехватывая функций `malloc` и `free` реализует собственную систему управления динамической памяти (кучу)[5]. Память для кучи может выделяться как стандартной функцией `malloc`, так и с помощью вызова `mmap`. Программа поддерживает выделение блоков размером в диапазоне от «ГР» до «МР*ГР», где МР – номер месяца Вашего рождения, ГР – год Вашего рождения. Выделение блоков других размеров недопустимо. Максимальный размер кучи задается параметром командной строки. Блоки выделяются по принципу «первый подходящий». Свободные блоки хранятся в виде одного двунаправленного несортированного списка.

Контрольные вопросы

1. Что такое «динамическое распределение памяти»? Чем оно отличается от «статического распределения памяти»?
2. Что такое «куча» (два смысла термина)?
3. В чем отличие областей brk/sbrk и mmap?
4. Какой алгоритм поиска свободных блоков использует функция malloc?
5. Что происходит при освобождении блока?

[5] Обратите внимание, что в программе использование функций стандартного ввода/вывода ограничено!!! (подробнее об этом рассказывается на лекции).