Тема 6. Управление памятью в GNU Linux

Главные вопросы, которые мы обсудим, представлены на СЛАЙДЕ 1. По ходу лекции даются примеры системных вызовов для распределения и освобождения областей памяти.

6.1 Адресное пространство процесса

Память — это один из ключевых ресурсов, доступных для использования процессами. В этом разделе мы покажем как управлять памятью, т.е. как ее распределять, манипулировать ею, а затем высвобождать. В современных системах основные сложности работы с памятью связаны не с тем, чтобы удовлетворить потребности всех нуждающихся, располагая небольшим объемом памяти, а с тем, чтобы правильно использовать распределенный регион памяти и отслеживать процесс такого использования. Мы рассмотрим некоторые из возможных способов распределения памяти в различных областях программы, разберем достоинства и недостатки этих методов. Мы также изучим некоторые способы установки содержимого произвольно выбранных областей памяти и оперирования этим содержимым. Мы научимся блокировать некоторый объем памяти, чтобы он не покидал пределов оперативной памяти, и программа не теряла работоспособность. При этом программе не придется дожидаться, пока ядро скопирует данные из области подкачки.

Linux, как и все современные ОС, виртуализирует имеющуюся в распоряжении физическую память. Процессы не обращаются непосредственно к физической памяти. Вместо этого ядро связывает каждый процесс с собственным уникальным **виртуальным адресным пространством** данного процесса (далее ВАП). Это ВАП является линейным, или плоским. Это значит, что его адреса начинаются с нуля, непрерывно увеличиваясь вплоть до заданного максимального значения. Кроме того, оно существует в единой области, непосредственно доступной и не требующей сегментирования.

При управлении памятью наиболее важной концепцией является страница. **Страница** — это наименьшая адресуемая сущность в памяти, которой может управлять блок управления памятью (MMU) процессора, поэтому можно сказать, что страницы «нарезаются» из адресного пространства процесса. Размер страницы зависит от применяемой машинной архитектуры. Наиболее распространенными размерами являются 4 Кбайт (в 32-битных системах) и 8 Кбайт (в 64-битных системах). Это всегда можно проверить следующим системным вызовом, который возвращает размер страницы в байтах:

```
#include <unistd.h>
int getpagesize (void);
```

Например:

```
int page_size = getpagesize ();
```

Процесс не обязательно будет обращаться ко всем страницам; конкретная страница может ничему не соответствовать. Страницы могут быть **валидными** либо

невалидными. Валидная страница ассоциирована с имеющейся страницей данных, которая располагается либо в физической памяти, либо в разделе подкачки или в файле на диске. Невалидная страница ни с чем не ассоциирована и представляет собой неиспользуемый, нераспределенный регион ВАП. При обращении к невалидной странице происходит нарушение сегментации.

Если валидная страница ассоциирована с данными, расположенными на вторичном носителе, то процесс не может получить доступ к этой странице, пока информация не будет перенесена в физическую память. Когда процесс пытается обратиться к такой странице, блок управления памятью генерирует страничное прерывание. Здесь в дело вступает ядро, прозрачно «**подкачивая**» данные в физическую память. Объем виртуальной памяти обычно значительно превышает объем физической памяти, поэтому ядро может выгрузить данные из памяти, освободив таким образом пространство для подкачки. **Выгрузка** — это перенос данных из физической памяти на вторичный носитель. Чтобы снизить количество последующих операций подкачки, ядро пытается выгрузить из памяти данные, которые с наименьшей вероятностью будут использоваться в ближайшем будущем.

Множественные страницы виртуальной памяти могут быть ассоциированы с единственной физической страницей, даже если они относятся к разным процессам, каждый из которых имеет собственное ВАП. Таким образом, различные ВАП могут совместно использовать (**разделять**) данные из физической памяти. Например, в любой момент вполне вероятно, что многие процессы системы совместно используют стандартную библиотеку языка С. При использовании **разделяемой памяти** каждый из этих процессов может отображать эту библиотеку на свое виртуальное адресное пространство, но в физической памяти при этом должна присутствовать лишь одна копия библиотеки. Еще один пример: два процесса могут одновременно отображать в память большую базу данных. В то время как нужная база данных будет присутствовать в виртуальном адресном пространстве каждого из этих процессов, в оперативной памяти будет находиться всего одна копия этой базы данных.

Разделяемые данные могут быть доступны только для чтения, записи или одновременно для того и другого. Когда процесс записывает информацию на совместно используемую страницу, допускающую такую запись, происходит одно из двух.

В простом случае ядро разрешает запись, после чего все процессы, использующие эту страницу, могут видеть результат записи. Как правило, если мы разрешаем множеству процессов считывать одну и ту же страницу или записывать на нее информацию, требуется обеспечить некоторую степень координации и синхронизации между этими процессами, но на уровне ядра запись «просто работает», и все процессы, разделяющие данные, немедленно видят все происходящие изменения.

Во втором случае блок управления памятью перехватывает операцию записи и генерирует исключительную ситуацию. В ответ ядро прозрачно создает новую копию страницы для записывающего процесса и позволяет процессу продолжать запись информации уже на новой странице. Такой подход называется «**копированием при записи**». Фактически процессы получают к разделяемым данным доступ на чтение, как следствие, экономится пространство памяти. Однако если процесс пытается записать информацию на разделяемую страницу, то получает уникальную копию этой страницы. Таким образом, ядро всегда может работать с учетом наличия собственной копии страницы у каждого записывающего процесса. Копирование при записи происходит постранично, поэтому такой прием фактически позволяет разделять огромный по размеру файл между многими процессами, и отдельные процессы будут получать уникальные физические копии только таких страниц, на которые они сами записывают информацию.

Ядро распределяет страницы по блокам, которые имеют набор тех или иных общих свойств — например, прав доступа. Эти блоки именуются **отображениями**, **сегментами** или **областями** памяти. Некоторые области памяти присутствуют в любом процессе.

- **текстовый сегмент** содержит программный код процесса, строковые литералы, константные значения переменных и другие данные, предназначенные только для чтения. В Linux этот сегмент обозначается доступным только для чтения и отображается прямо из двоичного файла (это может быть исполняемый файл программы или библиотека).

- **стековый сегмент**, как ни парадоксально, содержит стек выполнения процесса. Он может динамически расширяться или сжиматься по мере увеличения или уменьшения глубины стека. В стеке содержатся локальные (автоматические) переменные и возвращаемые данные функций. В многопоточном процессе каждому потоку соответствует собственный стек.

- **сегмент данных**, или **куча**, содержит динамическую память процесса. Этот сегмент доступен для записи, может расширяться или сжиматься. Вызов *malloc*() может удовлетворять запросы памяти из этого сегмента.

- **сегмент BSS** (*block started by symbol*) содержит неинициализированные глобальные переменные. В этих переменных находятся специальные значения (как правило, только нули) в соответствии со стандартом языка С.

Linux оптимизирует эти переменные двумя способами. Во-первых, поскольку BSS-сегмент предназначен для неинициализированных данных, компоновщик не сохраняет специальные значения в объектном файле, поэтому размер двоичного файла уменьшается. Во-вторых, когда этот сегмент загружается в память, ядро просто отображает его по принципу копирования при записи на страницу нулей, фактически устанавливая переменные в значения, которые заданы для них по умолчанию.

6.2 Распределение динамической памяти

Память программам предоставляется в форме автоматических и статических переменных, однако во основе любой подсистемы управления памятью лежат операции распределения, использования и возврата **динамической памяти**. Динамическая память возвращается во время выполнения, а не во время компиляции. При этом ее размер может быть неизвестен вплоть до момента выделения. Разработчик прибегает к использованию динамической памяти, когда объем требуемой памяти или длительность ее использования может варьироваться, и точные значения размера и длительности становятся известны только во время выполнения. Например, нам захотелось сохранить в памяти содержимое некоторого файла или пользовательского ввода, полученного с клавиатуры. Размер файла точно не известен, а пользователь может ввести много чего, поэтому размер буфера будет меняться, и нам понадобится его динамически увеличивать по мере считывания все большего количества данных.

В языке С нет переменной, в основе которой лежала бы динамическая память. Так, в С отсутствует механизм получения структуры *struct student*, существующей в динамической памяти. Вместо этого С предоставляет возможность выделения динамической памяти, достаточной для содержания структуры *student*. После этого программа будет взаимодействовать с памятью с помощью указателя; у нас *struct student* *.

Классический для языка С способ распределения динамической памяти называется *malloc*():

```
#include <stdlib.h>
void* malloc(size_t size);
```

При успешном вызове *malloc*() выделяет *size* байт памяти и возвращает указатель в начальную точку только что выделенной области. Содержимое памяти не определено, поэтому мы не должны рассчитывать, что память будет заполнена нулями. При ошибке *malloc*() возвращает NULL, а значение глобальной переменной *errno* устанавливается значение ENOMEM.

Использовать *malloc*() достаточно просто, как в следующем примере:

```
char *p;

/* дайте мне 3 Кбайт! */
p = malloc(3072);
if (NULL == p)
  perror("malloc");
```

В следующем примере память выделяется под структуру:

```
struct mind_map* map;
/*
* выделяем достаточное количество памяти для содержания структуры
* mind_map
* и указываем на нее с помощью 'map'
*/
map = malloc(sizeof (struct mind_map));
if (NULL == map)
  perror("malloc");
```

Язык C автоматически приводит указатели на *void* при любых типах назначения. Поэтому в наших примерах мы обходимся без приведения типа возвращаемого значения *malloc*() к типу *L*-значения, используемому при присваиваниях. Однако в языке C++ не выполняется автоматического приведения указателя на *void*, поэтому программисты, имеющие дело с C++, должны приводить тип возвращаемого значения *malloc*() так, как показано в следующем примере:

```
char* name;
/* выделяем 256 байт */
name = (char *)malloc(256);
if (NULL == name)
  perror("malloc");
```

Функция *malloc*() может возвращать NULL, поэтому разработчик просто обязан всегда проверять это условие и обрабатывать ошибки при их наличии.

Динамическое распределение памяти также может быть достаточно сложным, если размер *size* сам по себе является динамическим. Примером является динамическое распределение памяти для массивов, где размер элемента может быть фиксированным, а количество выделяемых элементов изменяется. Для упрощения данного сценария в библиотеке C предоставляется функция *calloc*():

```
#include <stdlib.h>
void* calloc(size_t num, size_t size);
```

При успешном вызове *calloc*() возвращает указатель на блок памяти, подходящий для содержания массива из *num* элементов, каждый из которых имеет размер *size* байт. Значит, объем памяти, запрашиваемый при этих следующих вызовах, идентичен (каждый вызов может вернуть больше памяти, чем первоначально запрашивал, но не меньше):

```
int* x;
int* y;
x = malloc(50 * sizeof (int));
if (NULL == x)
{
  perror("malloc");
  return -1;
}
y = calloc(50, sizeof (int));
```

```
if (NULL == y)
{
  perror("calloc");
  return -1;
}
```

Однако их поведение не является идентичным. В отличие от *malloc*(), функция *calloc*() заполняет нулями все байты в возвращаемом фрагменте памяти. Таким образом, каждый из 50 элементов, содержащихся в массиве целых чисел *y*, имеет значение 0, а содержимое элементов в *x* остается неопределенным. Если программа не собирается сразу же установить все 50 значений, то программист должен сам гарантировать, что элементы в массиве не будут заполнены мусором.

При ошибке *calloc*() и *malloc*() возвращают NULL и присваивают переменной *errno* значение ENOMEM.

Стандартная библиотека языка C предоставляет для изменения размера уже имеющихся распределенных областей следующую функцию:

```
#include <stdlib.h>
void* realloc(void *ptr, size_t size);
```

При успешном вызове *realloc*() размер области памяти, на которую направлен указатель *ptr*, изменяется. Новый размер в байтах задается через *size*. Она возвращает указатель на заново распределенную область памяти, причем этот указатель может быть как равен *ptr*, так и иметь другое значение. В случае увеличения области памяти *realloc*() может и не увеличить исходный фрагмент до запрашиваемого размера на том же месте, где этот фрагмент в данный момент находится. В таком случае функция попытается распределить новую область памяти размером *size* байт, скопировать старую область в новую, а старую после этого освободить. При любой подобной операции содержимое области памяти сохраняется либо полностью, либо в размере, равном объему новой выделенной области. Поскольку операции *realloc*() связаны с копированием, при увеличении области памяти они могут быть затратными.

Если размер *size* равен нулю, то эффект равносилен вызову *free*() применительно к *ptr*.

Если *ptr* равен NULL, результат операции аналогичен «свежему» использованию *malloc*(). Если указатель *ptr* не равен нулю, то он обязательно должен быть возвращен в предыдущем вызове, направленном к *malloc*(), *calloc*() или *realloc*().

При ошибке *realloc*() возвращает NULL и присваивает переменной *errno* значение ENOMEM. Состояние области памяти, на которую указывает *ptr*, остается неизменным.

Рассмотрим пример с уменьшением области памяти. Сначала воспользуемся *calloc*() и выделим достаточно памяти, чтобы содержать в ней двухэлементный массив структур *mind_map*:

```
struct mind_map* p;

/* выделяем память для двух структур mind_map */
p = calloc(2, sizeof (struct mind_map));
if (NULL == p)
{
  perror("calloc");
  return -1;
}

/* используем p[0] и p[1]... */
```

Теперь предположим, что одну мысль мы уже нашли, и вторая карта (*mind_map*) нам

больше не нужна. Тогда мы изменим занятую область памяти и отдадим ее половину, отведенную под карты, обратно в распоряжение системы. Такая операция не всегда целесообразна, но она полезна, если структура *mind_map* очень велика, а оставшуюся карту мы планируем хранить еще достаточно долго:

```
struct mind_map* r;

/* теперь нам нужна память для хранения только одной карты */
r = realloc(p, sizeof (struct mind_map));
if (NULL == r)
{
  /* NB: значение 'p' по-прежнему допустимо */
  perror("realloc");
  return -1;
}

/* используем 'r'... */
free(r);
```

У нас *p*[0] сохраняется после вызова *realloc*(). Что бы там ни было, оно остается доступным. Если вызов выполнится с ошибкой, то *p* остается нетронутым, а значит, действующим. Мы можем продолжать им пользоваться, а по окончании работы — освободить. И наоборот, если вызов завершится успешно, мы игнорируем *p*, а вместо него используем *r*. Теперь мы отвечаем за освобождение *r* после завершения работы.

6.3 Освобождение динамической памяти

В отличие от автоматически распределяемых областей, которые собираются системой без нашего участия, как только «распадается» стек, динамически распределенные области остаются неотъемлемыми частями ВАП, пока не будут высвобождены вручную. По этой причине программист отвечает за возврат динамически распределенной памяти в систему. Разумеется, все области памяти — распределенные статическим либо динамическим образом, высвобождаются автоматически, как только завершается весь процесс.

Память, распределенная с помощью *malloc*(), *calloc*() или *realloc*(), должна быть возвращена в систему, когда она больше не используется. Это делается с помощью *free*():

```
#include <stdlib.h>
void free(void* ptr);
```

Этот вызов освобождает память, на которую указывает *ptr*. Параметр *ptr* должен предварительно получен через *malloc*(), *calloc*() или *realloc*(). Это означает, что с помощью *free*() мы не сможем освобождать произвольные блоки памяти. Так, не получится освободить половину области памяти, передав указатель на ее середину. Такое действие приведет к появлению неопределенной памяти, что проявится в виде сбоя (**ошибки сегментации**).

Указатель *ptr* может быть равен NULL, в случае чего *free*() возвращает значение, поэтому распространенная практика проверки *ptr* на NULL перед вызовом *free*() не имеет смысла.

Приведем пример:

```
void print_chars(int n, char c)
{
  int i;
  for (i = 0; i < n; ++i)
  {
    char* s;
    int j;
```

```
    /*
     * Выделяем и заполняем нулями массив элементов i + 2
     * состоящий из символов. Обратите внимание: 'sizeof (char)'
     * всегда равно 1.
     */
    s = calloc(i + 2, 1);
    if (MULL == s)
    {
      perror("calloc");
      break;
    }

    for (j = 0; j < i + 1; ++j)
      s[j] = c;
    printf("%s\n", s);

    /* Все сделано. Можно вернуть память. */
    free(s);
  }
}
```

В этом примере мы выделяем *n* массивов из символов *char*, содержащих последовательно возрастающие количества элементов, начиная от двух байт до *n* + 1 элементов (*n* + 1 байт). Затем, обрабатывая каждый массив, цикл записывает символ *c* в каждый байт, кроме последнего (оставляя 0, уже находящийся в последнем байте). Массив выводится на экран как строка, после чего динамически выделенная память высвобождается.

Разумеется, есть и более эффективные способы реализации этой функции. Однако суть в том, что мы можем динамически распределять и высвобождать память, даже если размер и количество выделяемых областей становятся известны только во время выполнения.

Каковы же будут последствия, если в нашем примере не вызвать *free*(). Программа так и не вернет память в систему. Хуже того, она потеряет свою единственную ссылку на эту память — указатель *s*, в результате чего и сама больше не сможет обратиться к памяти. Такой тип программной ошибки называется **утечкой памяти**. Утечки и другие подобные ошибки, связанные с динамической памятью, являются одними из самых распространенных и, к сожалению, относятся к наиболее серьезным неполадкам при программировании на C. В языке C вся ответственность за управление памятью лежит на программисте, поэтому он должен очень внимательно следить за операциями распределения памяти.

Еще один распространенный подводный камень при программировании на C — это так называемое **использование освобожденной памяти**. Такая уязвимость возникает, если программа обращается к блоку памяти, который уже был освобожден. Как только для блока памяти вызвана функция *free*(), программа ни в коем случае не должна больше обращаться к содержимому этого блока. Программист должен проявлять особое внимание, отслеживая повисшие указатели (**висячие ссылки**): ненулевые, которые тем не менее указывают на недействительный блок памяти. Хороший инструмент, помогающий находить в программе ошибки, связанные с неверным использованием памяти, называется *Valgrind*. Его мы упомянули в предыдущем разделе.

6.4 Распределение памяти на основе стека

До сих пор мы обсуждали механизмы распределения динамической памяти, которые использовали для получения памяти кучу. Это неудивительно, поскольку куча является динамической по своей природе. В ВАП обычно находится еще одна программная конструкция — стек, в котором располагаются автоматические переменные программы.

При желании программист может использовать стек для распределения динамической памяти. Пока такое распределение не вызывает переполнения стека, подобный подход остается простым и осуществляется без проблем. Чтобы выполнить выделение динамической памяти из стека, мы воспользуемся системным вызовом *alloca*():

```
#include <alloca.h>
void* alloca(size_t size);
```

В случае успеха вызов *alloca*() возвращает указатель на *size* байт в памяти. Эта память находится в стеке и автоматически высвобождается после возврата функции, которая инициировала вызов. Некоторые реализации при ошибке возвращают NULL, но большинство реализаций *alloca*() не реагируют на ошибки и, соответственно, не могут о них сообщать. Сбой проявляется только на этапе переполнения стека.

Эта функция используется идентично *malloc*(), но не требуется высвобождать распределенную память. Ниже приведен пример функции, которая открывает указанный файл в конфигурационном каталоге системы. Скорее всего таким будет каталог /*etc*, — который машинонезависимо определяется во время компиляции. Функция должна распределить место для нового буфера, скопировать в него конфигурационный каталог системы, а потом сцепить этот буфер с предоставленным именем файла:

```
int open_sysconf(const char* file, int flags, int mode)
{
  const char* etc = SYSCONF_DIR;   /* "/etc/" */
  char* name;
  name = alloca(strlen(etc) + strlen(file) + 1);
  strcpy(name, etc);
  strcat(name, file);
  return open(name, flags, mode);
}
```

После возврата память, выделенная с помощью функции *alloca*(), автоматически высвобождается, поскольку стек возвращается к вызывающей функции. Таким образом, после возврата функции, вызвавшей *alloca*(), мы уже не сможем использовать эту память. Тем не менее, поскольку нам не приходится выполнять какую-либо очистку, т.е. вызывать *free*(), то результирующий код получается немного короче. Вот та же функция, реализованная с помощью *malloc*():

```
int open_sysconf(const char* file, int flags, int mode)
{
  const char* etc = SYSCONF_DIR; /* "/etc/" */
  char* name;
  int fd;
  name = malloc(strlen(etc) + strlen(file) + 1);
  if (NULL == name)
  {
    perror("malloc");
    return -1;
  }
  strcpy(name, etc);
  strcat(name, file);
  fd = open(name, flags, mode);
  free(name);
  return fd;
}
```

Не следует использовать память, выделенную с помощью *alloca*(), в параметрах, передаваемых вызову функции, поскольку в таком случае распределенная память окажется где-то в середине стекового пространства, зарезервированного для параметров функций. Например, следующий код неверен:

```
/* НЕЛЬЗЯ ДЕЛАТЬ ТАК! */
ret = foo(x, alloca(10));
```

Итак, если мы стремимся обеспечить переносимость нашего кода, то не должны использовать *alloca*(). Тем не менее в рамках Linux *alloca*() является очень неплохим инструментом. Он работает исключительно хорошо — на многих архитектурах при распределении памяти с помощью *alloca*() приходится всего лишь увеличить указатель стека. Этот вызов оказывается удобнее и производительнее, чем *malloc*(). При выделении небольших объемов памяти в уникальном для Linux коде *alloca*() позволяет значительно улучшить рабочие характеристики системы.

Один из самых распространенных случаев использования *alloca*() связан с временным дублированием строки. Например:

```
/* мы хотим дублировать 'song' */
char* dup;
dup = alloca(strlen(song) + 1);
strcpy(dup, song);

/* используем 'dup'... */
return; /* 'dup' автоматически высвобождается */
```

Такая операция требуется не так уж редко, а также, учитывая существенное ускорение работы при применении *alloca*(), в Linux-системах предоставляются варианты *strdup*(), дублирующие указанную строку в стеке:

```
#define _GNU_SOURCE
#include <string.h>
char* strdupa(const char* s);
char* strndupa(const char* s, size_t n);
```

Вызов *strdupa*() возвращает дубликат *s*. При вызове *strndupa*() дублируется до *n* символов из *s*. Если *s* длиннее *n*, то дублирование прекращается на *n*, и функция прикрепляет нулевой байт. Эти функции обладают всеми достоинствами *alloca*(). Дублированная строка автоматически высвобождается, как только завершается функция, сделавшая вызов.

В C99 появились **массивы переменной длины** (VLA), «геометрия» которых задается во время выполнения, а не компиляции. Массивы переменной длины позволяют избежать издержек, связанных с выделением динамической памяти, почти по такому же принципу, что и *alloca*().

Они работают примерно так, как мы могли бы предположить:

```
for (i = 0; i < n; ++i)
{
  char foo[i + 1];
  /* используем 'foo'... */
}
```

Здесь *foo* — это массив символов *char*, имеющий переменный размер $i + 1$. На каждой итерации цикла *foo* создается динамически и автоматически освобождается, как только он оказывается за пределами области определения. Если вместо VLA мы воспользуемся *alloca*(), то память не будет освобождена вплоть до возврата функции. Массив переменной длины гарантирует, что освобождение памяти происходит при каждой итерации цикла. Соответственно, используя массив переменной длины, мы потребим не более n байт, тогда как *alloca*() потребуется $n * (n + 1) / 2$ байт.

Применив массив переменной длины, мы можем еще раз переписать нашу функцию *open_sysconf*():

```
int open_sysconf(const char* file, int flags, int mode)
{
  const char* etc; = SYSCONF_DIR; /* "/etc/" */
  char name[strlen(etc) + strlen(file) + 1];
  strcpy(name, etc);
  strcat(name, file);
  return open(name, flags, mode);
}
```

Основное различие между *alloca*() и VLA заключается в следующем: память, полученная с помощью первой функции, существует на протяжении жизни функции, в то время как во втором случае память имеется в распоряжении до выхода содержащей ее переменной из области определения; а такой выход может наступить до возврата функции. Это может иметь как положительные, так и отрицательные последствия. В цикле *for*, который мы рассмотрели, освобождение памяти при каждой итерации цикла снижает общий объем потребляемой памяти без каких-либо побочных эффектов. Нам дополнительная память не нужна. Однако если мы по каким-то причинам захотим, чтобы память имелась в наличии дольше, чем на протяжении одного цикла, то целесообразно использовать *alloca*().

6.5 Управление сегментом данных

Исторически в системах UNIX предоставлялись интерфейсы для непосредственного управления сегментом данных. Тем не менее в большинстве программ эта возможность практически не находила применения, так как *malloc*() и другие способы выделения памяти значительно проще в использовании, а также мощнее. Мы опишем подобные интерфейсы для полноты картины, а также для тех, кто рискнет реализовать собственный механизм выделения памяти на основе работы с кучей:

```
#include <unistd.h>
int brk(void* end);
void* sbrk(intptr_t increment);
```

Названия этих функций происходят из старых UNIX-систем, в которых куча и стек еще находились в одном сегменте памяти. Распределение динамической памяти в куче выполнялось вверх, начиная с нижней части сегмента; стек рос вниз из верхней части сегмента, по направлению к куче. Граница, разделявшая области стека и кучи, называлась **остановом** или **точкой останова**. В современных системах, в которых сегмент данных находится в собственной части ВАП, мы все так же именуем ее конечный адрес точкой останова.

Вызов *brk*() устанавливает точку останова (конец сегмента данных), задавая адрес, указанный в *end*. В случае успеха функция возвращает 0. При ошибке она возвращает –1 и устанавливает *errno* значение ENOMEM.

Вызов *sbrk*() обеспечивает приращение конца сегмента данных на *increment* байт, причем как на положительное, так и на отрицательное значение. Функция *sbrk*() возвращает пересмотренную точку останова. Соответственно, приращение, равное 0, дает нам актуальную точку останова:

```
printf("Текущая точка останова — %p\n", sbrk(0));
```

Ни в стандарте C, ни в POSIX не определяется ни одна из этих функций. Однако практически во всех UNIX-системах поддерживается как минимум одна из них, а то и две. Переносимые программы должны работать только со стандартизированными интерфейсами.

6.6 Операции над областями памяти

В стандартной библиотеке языка С определено семейство функций для управления необработанными байтами памяти. По принципу работы эти функции во многом напоминают интерфейсы для манипуляций со строками, такие как *strcmp*() и *strcpy*(). Однако они взаимодействуют с буфером, размер которого задается пользователем, а не опираются на предположение, что строки завершаются нулем. Внимание: ни одна из этих функций не может возвращать ошибок. Предотвращение сбоев — задача, которую приходится решать программисту. Достаточно передать неверную область памяти — и не останется выхода, кроме как идти на вынужденное нарушение сегментирования!

Из функций для управления памятью чаще всего применяется *memset*():

```
#include <string.h>
void* memset(void* s, int c, size_t n);
```

Вызов *memset*() устанавливает *n* байт, начиная с *s*, в байт *c* и возвращает *s*. Часто эта функция используется для заполнения блока памяти нулями:

```
/* обнуляем [s,s+256) */
memset(s, '\0', 256);
```

Также Linux предоставляет функцию *bzero*() для обеспечения обратной совместимости и переносимости на другие системы:

```
#include <strings.h>
void bzero(void* s, size_t n);
```

Следующий вызов аналогичен приведенному выше примеру с *memset*():

```
bzero(s, 256);
```

Аналогично *strcmp*(), функция *memcmp*() проверяет два фрагмента памяти на эквивалентность:

```
#include <string.h>
int memcmp(const void* s1, const void* s2, size_t n);
```

Вызов сравнивает первые *n* байт *s*1 с первыми *n* байтами в *s*2. После этого возвращает 0, если блоки памяти эквивалентны. Если *s*1 меньше *s*2, то возвращается значение меньше 0, а если *s*1 больше *s*2, то возвращается значение больше 0.

Устаревший способ для этой же задачи:

```
#include <strings.h>
int bcmp(const void* s1, const void* s2, size_t n);
```

При вызове *bcmp*() сравниваются первые *n* байт в *s*1 и *s*2. Если блоки памяти эквивалентны, то возвращается нулевое значение, если неэквивалентны — ненулевое.

В силу применения структур данных, их сравнение на эквивалентность с помощью *memcmp*() или *bcmp*() — неточная операция. При заполнении может применяться неинициализированный мусор, состав которого у двух экземпляров структуры отличается, тогда как в остальном эти структуры идентичны. Следовательно, приведенный далее код является небезопасным:

```
/* идентичны ли две шлюпки? (НЕБЕЗОПАСНО) */
int compare_dinghies(struct dinghy* a, struct dinghy* b)
{
  return memcmp(a, b, sizeof (struct dinghy));
}
```

Если программист собирается сравнивать структуры, то нужно сравнивать друг с другом каждый элемент этих структур по очереди. Такой поход допускает некоторую оптимизацию, но определенно является более трудоемким, чем небезопасная работа с применением *memcmp*(). Вот эквивалентный код:

```
/* Идентичны ли две шлюпки? */
int compare_dinghies(struct dinghy* a, struct dinghy* b)
{
  int ret;
  if (a->nr_oars < b->nr_oars)
    return -1;
  if (a->nr_oars > b->nr_oars)
    return 1;
  ret = strcmp(a->boat_name, b->boat_name);
  if (NULL != ret)
    return ret;
  /* и т. д. для каждого элемента... */
}
```

Функция *memmove*() копирует первые *n* байт из *src* в *dst*, возвращая *dst*:

```
#include <string.h>
void* memmove(void* dst, const void* src, size_t n);
```

Устаревший аналог:

```
#include <strings.h>
void bcopy(const void* src, void* dst, size_t n);
```

Обе функции принимают одинаковые параметры, однако порядок первых двух в *bcopy*() является обратным.

И *bcopy*(), и *memmove*() позволяют безопасно обрабатывать пересекающиеся области памяти (например, если часть *dst* находится внутри *src*). Таким образом можно перемещать байты в определенной области памяти вверх-вниз. Данная ситуация встречается редко, а программисту необходимо знать о ее наступлении, поэтому стандарт C определяет вариант *memmove*(), не поддерживающий такого взаимного пересечения областей памяти. Этот вариант в большинстве случаев работает быстрее:

```
#include <string.h>
void* memcpy(void* dst, const void* src, size_t n);
```

Функция работает аналогично *memmove*(), с оговоркой, что пересечение *dst* и *src* не разрешается. Если это происходит, то результат не определен.

Еще одна безопасная функция для копирования называется *memccpy*():

```
#include <string.h>
void* memccpy (void* dst, const void* src, int c, size_t n);
```

Функция *memccpy*() работает так же, как и *memcpy*(), но копирование останавливается, если функция находит байт *c* среди первых *n* байт в *src*. Вызов возвращает указатель на байт, который следует в *dst* после *c*, либо NULL, если *c* не был найден.

Наконец, можно использовать *mempcpy*() для пошагового прохода через память:

```
#define _GNU_SOURCE
#include <string.h>
void* mempcpy(void* dst, const void* src, size_t n);
```

Функция *mempcpy*() работает так же, как и *memcpy*(), но возвращает указатель на

следующий байт после последнего скопированного байта. Она полезна, если множество данных требуется скопировать в последовательно расположенные области памяти. Однако улучшение незначительное, так как возвращаемое значение — это просто сумма *dst* + *n*. Эта функция является уникальной для GNU.

Функции *memchr*() и *memrchr*() находят заданный байт в блоке памяти:

```
#include <string.h>
void* memchr(const void* s, int c, size_t n);
```

Функция *memchr*() просматривает *n* байт в области памяти, на которую направлен указатель *s*, и ищет символ *c*, интерпретируемый как *unsigned char*:

```
#define _GNU_SOURCE
#include <string.h>
void* memrchr(const void* s, int c, size_t n);
```

Вызов возвращает указатель на первый байт, совпадающий с *c*, либо NULL, если *c* не найден Функция *memrchr*() равнозначна *memchr*(), но она выполняет поиск назад, начиная с байта *n*, на который указывает *s*, а не вперед — от нулевого байта. В отличие от *memchr*(), функция *memrchr*() — это расширение GNU, а не часть языка C.

Более сложные поисковые задачи решаются с помощью функции *memmem*(). Она ищет в блоке памяти произвольный массив байтов:

```
#define _GNU_SOURCE
#include <string.h>
void* memmem(const void* haystack,
  size_t haystacklen,
  const void* needle,
  size_t needlelen
);
```

Функция *memmem*() возвращает указатель на первый экземпляр подблока *needle*, имеющего длину *needlelen* байт, который, в свою очередь, находится в блоке памяти *haystack* длиной *haystacklen* байт. Если функции не удается найти *needle* в *haystack*, то она возвращает NULL. Эта функция также является расширением GNU.

Стандартная библиотека C в Linux предоставляет интерфейс для примитивного сворачивания байтов данных:

```
#define _GNU_SOURCE
#include <string.h>
void* memfrob(void* s, size_t n);
```

Вызов *memfrob*() затемняет первые *n* байт в памяти, начиная с *s*. При этом применяется исключающее «ИЛИ» (XOR) к каждому байту с числом 42. Вызов возвращает *s*.

Эффект от вызова *memfrob*() можно обратить, вновь вызвав *memfrob*() применительно к той же области памяти, поэтому следующий код является холостой командой, применяемой с *secret*:

```
memfrob(memfrob(secret, len), len);
```

Эта функция не является полноценной и даже приемлемой заменой шифрования; ее использование ограничено несложным затемнением строк. Функция является уникальной для GNU.

6.7 Блокирование областей памяти

В Linux реализуется **подкачка страниц по требованию**. В соответствии с этой

технологией страницы «подкачиваются» с диска по мере необходимости и сбрасываются на диск, как только становятся не нужны. Благодаря этому ВАП процессов, работающих в системе, могут никак не соотноситься с общим объемом физической памяти, т.к. вторичные носители способны создавать видимость наличия практически неограниченного объема физической памяти.

Такая подкачка происходит прозрачно, и наши приложения обычно вообще не должны задумываться о том, как подкачка организуется в ядре ОС. Однако возможны две ситуации, в которых приложению бывает целесообразно оказывать влияние на процедуры подкачки, происходящие в системе.

- **Детерминизм**. Приложения, работающие с ограничениями по времени, требуют детерминированного поведения. Если некоторые обращения к памяти приводят к возникновению страничных сбоев, связанных с затратными операциями ввода-вывода, то приложение может испытывать необходимость в дополнительном времени на работу. Если приложение может гарантировать, что необходимые ему страницы всегда остаются в физической памяти и не сбрасываются на диск, то оно может обеспечить и отсутствие страничных сбоев при обращениях к памяти. Так достигается согласованность и детерминизм, а также улучшается производительность.

- **Безопасность**. Если в памяти хранятся конфиденциальные сведения, их можно оттуда выгрузить и хранить на диске в незашифрованном виде. Например, если закрытый ключ пользователя обычно хранится на диске в зашифрованном виде, то незашифрованная копия ключа, находившаяся в памяти, может оказаться в файле подкачки. В среде, где действуют повышенные требования к безопасности, такое поведение может оказаться неприемлемым. Если в приложении возникает подобная проблема, оно может потребовать, чтобы область, содержащая ключ, всегда оставалась в пределах физической памяти.

Разумеется, если изменить принципы подкачки, действующие в ядре, это может негативно сказаться на общей производительности. Детерминизм или безопасность в одном приложении могут улучшиться, но пока его страницы будут заблокированы в памяти, вместо них будут подкачиваться страницы другого приложения. Если мы доверяем алгоритмам ядра, то можем быть уверены, что оно оптимальным образом будет подбирать страницы для выгрузки из памяти. Иначе говоря, будет выгружать страницы, которые с наименьшей вероятностью понадобятся нам в ближайшем будущем, поэтому, если мы изменим данное поведение, ядро может выгрузить какие-то сравнительно востребованные страницы.

К вопросу блокирования областей памяти и системным вызовам, предназначенным для этого, мы вернемся в одном из следующих разделов.

Литература и дополнительные источники к Теме 6

1. Gorman, M. Understanding the Linux Virtual Memory Manager. - Pearson Education, 2004. – 768 pp.
2. Страничная память – http://ru.wikipedia.org/wiki/Страничная_память
3. Элементы архитектуры системы виртуальной памяти во FreeBSD - http://www.freebsd.org/doc/ru_RU.KOI8-R/articles/vm-design/index.html