6. Управление памятью в GNU Linux

Разделы:

- Адресное пространство процесса
- Распределение динамической памяти
- Освобождение динамической памяти
- Распределение памяти на основе стека
- Управление сегментом данных
- Операции над областями памяти
- Блокирование областей памяти

Адресное пространство процесса

- Linux 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)
- ВАП является линейным, или плоским
  - 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

Адресное пространство процесса

- **Страница** — это наименьшая адресуемая сущность в памяти, которой может управлять блок управления памятью (MMU) процессора, поэтому страницы «нарезаются» из адресного пространства процесса
- Размер страницы зависит от применяемой машинной архитектуры

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

int page_size = getpagesize ();
```

Адресное пространство процесса

- 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

Адресное пространство процесса

- Ядро распределяет страницы по блокам, которые имеют набор тех или иных общих свойств — например, прав доступа
- Эти блоки - отображения, сегменты или области
- Некоторые области (сегменты) памяти присутствуют в любом процессе

Адресное пространство процесса

- **текстовый сегмент** содержит программный код процесса, строковые литералы, константные значения переменных и другие данные, предназначенные только для чтения
- **стековый сегмент** содержит стек выполнения процесса
- **сегмент данных**, или **куча**, содержит динамическую память процесса
- **сегмент BSS** (*block started by symbol*) содержит неинициализированные глобальные переменные

Распределение динамической памяти

- Память программам предоставляется в форме автоматических и статических переменных, но в основе любой подсистемы управления памятью лежат операции распределения, использования и возврата динамической памяти
- Динамическая память возвращается во время выполнения, а не во время компиляции
- В языке 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* *

Распределение динамической памяти

```
#include <stdlib.h>
void* malloc(size_t size);
char *p;
/* дайте мне 3 Кбайт! */
p = malloc(3072);
if (NULL == p)
  perror("malloc");
-----------------------------------------
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");
```

Распределение динамической памяти

```
char* name;

/* выделяем 256 байт */
name = (char *)malloc(256);
if (NULL == name)
  perror("malloc");
-----------------------------
#include <stdlib.h>
void* calloc(size_t num, size_t size);
-----------------------------
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;
}
```

Распределение динамической памяти

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

Освобождение динамической памяти

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

Освобождение динамической памяти

- Каковы последствия, если не вызвать *free*()?
- Программа так и не вернет память в систему
- Это **утечка памяти**

- Как только для блока памяти вызвана функция *free*(), программа ни в коем случае не должна больше обращаться к содержимому этого блока
- Значит, повторный вызов *free*() может приводить к ошибкам, в том числе сегментации
- Ошибки детектируются инструментами типа *Valgrind*

Распределение памяти на основе стека

- В ВАП обычно находится такая программная конструкция, как стек, в котором располагаются автоматические переменные программы

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

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);
}
```

Распределение памяти на основе стека

```
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;
}
---------------------------------------
/* НЕЛЬЗЯ ДЕЛАТЬ ТАК! */
ret = foo(x, alloca(10));
---------------------------------------
/* мы хотим дублировать 'song' */
char* dup;
dup = alloca(strlen(song) + 1);
strcpy(dup, song);

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

Распределение памяти на основе стека

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

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

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);
}
```

Управление сегментом данных

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

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

Операции над областями памяти

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

/* обнуляем [s,s+256) */
memset(s, '\0', 256);
----------------------------------------
#include <strings.h>
void bzero(void* s, size_t n);


bzero (s, 256);
----------------------------------------
#include <string.h>
int memcmp(const void* s1, const void* s2, size_t n);

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

Операции над областями памяти

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

/* 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... */
}
```

Операции над областями памяти

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

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

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

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

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

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

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

Операции над областями памяти

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

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

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

Блокирование областей памяти

- В Linux реализуется **подкачка страниц по требованию**
- Возможны две ситуации, в которых приложению бывает целесообразно оказывать влияние на процедуры подкачки, происходящие в системе
  - Детерминизм
  - Безопасность
- Нужна блокировка областей в ОП
- Рассматривается в разделе «Прочие системные вызовы»

See also

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