http://rus-linux.net/MyLDP/BOOKS/Moduli-yadra-Linux/06/kern-mod-06-08.html

Динамические структуры и управление памятью

Статический и динамический способ размещения структур данных имеют свои положительные и отрицательные стороны, главными из которых принято считать: а). статическая память: надёжность, живучесть и меньшая подверженность ошибкам; б). динамическая память: гибкость использования. Использование динамических структур всегда требует того или иного механизма управления памятью: создание и уничтожение терминальных элементов динамически увязываемых структур.

Циклический двусвязный список

Чтобы уменьшить количество дублированного кода, разработчики ядра создали (с ядра 2.6) стандартную реализацию кругового, двойного связного списка; всем другим нуждающимся в манипулировании списками (даже простейшими линейными односвязными, к примеру) рекомендуется разработчиками использовать это средство. Именно поэтому они заслуживают отдельного рассмотрения.

Примечание: *При работе с интерфейсом связного списка всегда следует иметь в виду, что функции списка выполняют без блокировки. Если есть вероятность того, что драйвер может попытаться выполнить на одном списке конкурентные операции, вашей обязанностью является реализация схемы блокировки. Альтернативы (повреждённые структуры списка, потеря данных, паники ядра), как правило, трудно диагностировать.*

Чтобы использовать механизм списка, ваш драйвер должен подключить файл <linux/list.h>. Этот файл определяет простую структуру типа list_head:

```
struct list_head {
  struct list_head *next, *prev;
};
```

Для использования в вашем коде средства списка Linux, необходимо лишь вставлять list_head внутри собственных структур, входящих в список, например:

```
struct todo_struct {
  struct list_head list;
  int priority;
  /* ... добавить другие зависимые от задачи поля */
};
```

Заголовки списков должны быть проинициализированы перед использованием с помощью макроса INIT_LIST_HEAD. Заголовок списка может быть объявлен и проинициализирован так (динамически):

```
struct list_head todo_list;
INIT_LIST_HEAD( &todo_list );
```

Альтернативно, списки могут быть созданы и проинициализированы статически при компиляции:

```
LIST_HEAD( todo_list );
```

Некоторые функции для работы со списками определены в <linux/list.h>. Как мы видим, API работы с циклическим списком позволяет выразить любые операции с элементами списка, не вовлекая в операции манипулирование с внутренними полями связи списка; это очень ценно для сохранения целостности списков:

```
list_add( struct list_head *new, struct list_head *head );
```

- добавляет новую запись new сразу же после головы списка, как правило, в начало списка. Таким образом, она может быть использована для создания стеков. Однако, следует отметить, что голова не должна быть номинальной головой списка; если вы передадите структуру list_head, которая окажется где-то в середине списка, новая запись пойдёт сразу после неё. Так как списки Linux являются круговыми, голова списка обычно не отличается от любой другой записи.

```
list_add_tail( struct list_head *new, struct list_head *head );
```

- добавляет элемент new перед головой данного списка - в конец списка, другими словами, list_add_tail() может, таким образом, быть использована для создания очередей первый вошёл - первый вышел.

```
list_del( struct list_head *entry );
list_del_init( struct list_head *entry );
```

- данная запись удаляется из списка. Если эта запись может быть когда-либо вставленной в другой список, вы должны использовать list_del_init(), которая инициализирует заново указатели связного списка.

```
list_move( struct list_head *entry, struct list_head *head );
list_move_tail( struct list_head *entry, struct list_head *head );
```

- данная запись удаляется из своего текущего положения и перемещается (запись) в начало голову списка. Чтобы переместить запись в конец списка используется list_move_tail().

```
list_empty( struct list_head *head );
```

- возвращает ненулевое значение, если данный список пуст.

```
list_splice( struct list_head *list, struct list_head *head );
```

- объединение двух списков вставкой нового списка list сразу после головы head.

Структуры list_head хороши для реализации линейных списков, но использующие его программы часто больше заинтересованы в некоторых более крупных структурах, которые увязываются в список как целое. Предусмотрен макрос list_entry, который связывает указатель структуры list_head обратно с указателем на структуру, которая его содержит. Он вызывается следующим образом:

```
list_entry( struct list_head *ptr, type_of_struct, field_name );
```

- где ptr является указателем на используемую структуру list_head, type_of_struct является типом структуры, содержащей этот ptr, и field_name является именем поля списка в этой структуре.

Пример: в нашей ранее показанной структуре todo_struct поле списка называется просто list. Таким образом, мы бы хотели превратить запись в списке listptr в соответствующую структуру, то могли бы выразить это такой строкой:

```
struct todo_struct *todo_ptr = list_entry( listptr, struct todo_struct, list );
```

Макрос list_entry() несколько необычен и требует некоторого времени, чтобы привыкнуть, но его не так сложно использовать.

Обход связных списков достаточно прост: надо только использовать указатели prev и next. В качестве примера предположим, что мы хотим сохранить список объектов todo_struct, отсортированный в порядке убывания. Функция добавления новой записи будет выглядеть примерно следующим образом:

```
void todo_add_entry( struct todo_struct *new ) {
```

```
  struct list_head *ptr;
  struct todo_struct *entry;
  /* голова списка поиска: todo_list */
  for( ptr = todo_list.next; ptr != &todo_list; ptr = ptr->next ) {
    entry = list_entry( ptr, struct todo_struct, list );
    if( entry->priority < new->priority ) {
      list_add_tail( &new->list, ptr);
      return;
    }
  }
  list_add_tail( &new->list, &todo_list );
}
```

Однако, как правило, лучше использовать один из набора предопределённых макросов для создание циклов, которые перебирают списки. Например, предыдущий цикл мог бы быть написать так:

```
void todo_add_entry( struct todo_struct *new ) {
  struct list_head *ptr;
  struct todo_struct *entry;
  list_for_each( ptr, &todo_list ) {
    entry = list_entry( ptr, struct todo_struct, list );
    if( entry->priority < new->priority) {
      list_add_tail( &new->list, ptr );
      return;
    }
  }
  list_add_tail( &new->list, &todo_list );
}
```

Использование предусмотренных макросов помогает избежать простых ошибок программирования; разработчики этих макросов также приложили некоторые усилия, чтобы они выполнялись производительно. Существует несколько вариантов:

```
list_for_each( struct list_head *cursor, struct list_head *list )
```

- макрос создаёт цикл for, который выполняется по одному разу с указателем cursor, присвоенным поочерёдно указателю на каждую последовательную позицию в списке (будьте осторожны с изменением списка при итерациях через него).

```
list_for_each_prev( struct list_head *cursor, struct list_head *list )
```

- эта версия выполняет итерации назад по списку.

```
list_for_each_safe( struct list_head *cursor, struct list_head *next, struct list_head *list
)
```

- если операции в цикле могут удалить запись в списке, используйте эту версию: он просто сохраняет следующую запись в списке в next для продолжения цикла, поэтому не запутается, если запись, на которую указывает cursor, удаляется.

```
list_for_each_entry( type *cursor, struct list_head *list, member )
list_for_each_entry_safe( type *cursor, type *next, struct list_head *list, member )
```

- эти **макросы** облегчают процесс просмотр списка, содержащего структуры данного типа type. Здесь cursor является указателем на содержащий структуру тип, и member является именем структуры list_head внутри содержащей структуры. С этими макросами нет необходимости помещать внутри цикла вызов макроса list_entry().

В заголовках <linux/list.h> определены ещё некоторые дополнительные декларации для описания динамических структур.

Модуль использующий динамические структуры

Ниже показан пример модуля ядра (архив list.tgz), строящий, использующий и утилизирующий простейшую динамическую структуру в виде односвязного списка:

mod_list.c :

```
#include <linux/slab.h>
#include <linux/list.h>
MODULE_LICENSE( "GPL" );
static int size = 5;
module_param( size, int, S_IRUGO | S_IWUSR );     // размер списка как параметр модуля
struct data {
    int n;
    struct list_head list;
};
void test_lists( void ) {
  struct list_head *iter, *iter_safe;
  struct data *item;
  int i;
  LIST_HEAD( list );
  for( i = 0; i < size; i++ ) {
    item = kmalloc( sizeof(*item), GFP_KERNEL );
    if( !item ) goto out;
    item->n = i;
    list_add( &(item->list), &list );
  }
  list_for_each( iter, &list ) {
    item = list_entry( iter, struct data, list );
    printk( KERN_INFO "[LIST] %d\n", item->n );
  }
out:
  list_for_each_safe( iter, iter_safe, &list ) {
    item = list_entry( iter, struct data, list );
    list_del( iter );
    kfree( item );
  }
}
static int __init mod_init( void ) {
  test_lists();
  return -1;
}
>module_init( mod_init );
```

```
$ sudo /sbin/insmod ./mod_list.ko size=3
insmod: error inserting './mod_list.ko': -1 Operation not permitted
$ dmesg | tail -n3
[LIST] 2
[LIST] 1
[LIST] 0
```

Сложно структурированные данные

Одной только ограниченной структуры данных struct list_head достаточно для построения динамических структур практически произвольной степени сложности, как, например, сбалансированные B-деревья, красно-чёрные списки и другие. Именно поэтому ядро 2.6 было полностью переписано в части используемых списковых структур на использование struct list_head. Вот каким простым образом может быть представлено с использованием этих структур бинарное дерево:

```
struct my_tree {
  struct list_head left, right; /* левое и правое поддеревья */
  /* ... добавить другие зависимые от задачи поля */
};
```

Не представляет слишком большого труда для такого представления создать собственный набор функций его создания-инициализации и манипуляций с узлами такого дерева.

Обсуждение

При обсуждении заголовков списков, было показано две (альтернативно, на выбор) возможности объявить и инициализировать такой заголовок списка:

- статический (переменная объявляется макросом и тут же делаются все необходимые для инициализации манипуляции):

```
LIST_HEAD( todo_list );
```

- динамический (переменная сначала объявляется, как и любая переменная элементарного типа, например, целочисленного, а только потом инициализируется указанием её адреса):

```
struct list_head todo_list;
INIT_LIST_HEAD( &todo_list );
```

Такая же дуальность (статика + динамика) возможностей будет наблюдаться далее много раз, например относительно всех примитивов синхронизации. Напрашивается вопрос: зачем такая избыточность возможностей и когда что применять? Дело в том, что очень часто (в большинстве случаев) такие переменные не фигурируют в коде автономно, а встраиваются в более сложные объемлющие структуры данных. Вот для таких встроенных объявлений и будет годиться только динамический способ инициализации. Единичные переменные проще создавать статически.