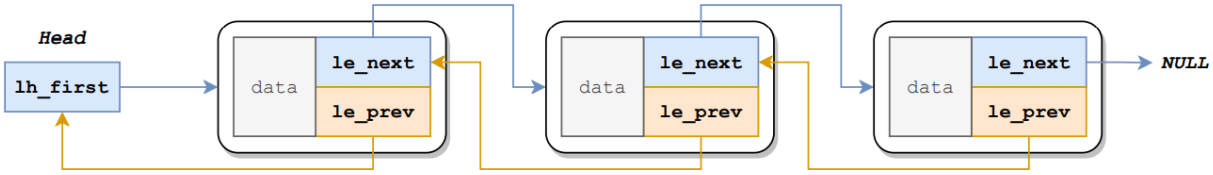
队列宏



使用

链表

创建与初始化

```
//定义链表指针项，由于LIST_ENTRY()中定义的是无名结构体，所以必须使用typedef
//Page : 链表项结构体类型
//Page_LIST_entry_t : 链表项结构体中包含的结构体，本身包含了指向下一项的指针，和指向上一项下一项指针的指针
typedef LIST_ENTRY(Page) Page_LIST_entry_t;

//定义链表项结构体
//pp_link : 包含了上述指针的结构体变量名，在使用时作为 field 变量传给宏
struct Page {
    Page_LIST_entry_t pp_link;
    //所需要的其他数据
}

//定义链表表头
//Page_list : 链表表头结构体类型，在宏内部使用有名结构体定义方法，使用时可以如 struct Page_list
LIST_HEAD(Page_list, Page);

//创建链表
struct Page_list page_free_list;

//初始化链表头，记得要传入指针
LIST_INIT(&page_free_list);
```

插入与删除

```
//插入新的元素
//listelm : 准备在其后插入新元素的链表中元素
//elm : 待插入元素
//field : 链表元素结构体中，保存指针的结构体变量名。如pp_link
LIST_INSERT_AFTER(listelm, elm, field);
LIST_INSERT_BEFORE(listelm, elm, field);
LIST_INSERT_HEAD(head, elm, field);

//elm : 待删除的表中元素
LIST_REMOVE(elm, field)
```

其它

```
#define LIST_NEXT(elm, field) ((elm)->field.le_next)
#define LIST_EMPTY(head) ((head)->lh_first == NULL)
#define LIST_FIRST(head) ((head)->lh_first)
```

遍历

```
//这是一个for循环头，使用的时候后面直接接大括号，var是指向元素的指针
#define LIST_FOREACH(var, head, field) \
        for ((var) = LIST_FIRST((head)); (var); (var) = LIST_NEXT((var), field))
```

尾队列

创建与初始化

```
//定义元素结构体
struct Env {
        LIST_ENTRY(Env) env_link; // Free list
        TAILQ_ENTRY(Env) env_sched_link;
        //...
};
//定义表头
TAILQ_HEAD(Env_sched_list, Env);
//声明表头变量
struct Env_sched_list env_sched_list;
```

使用

```
//从头部插入
TAILQ_INSERT_HEAD(&env_sched_list, e, env_sched_link);

// 不会改变elm本身的内容
TAILQ_REMOVE(head, elm, field);

TAILQ_FIRST(head) ((head)->tqh_first);

// 这个宏仅使用elm中的值更新队列中的值，连续重复调用不会产生错误
TAILQ_INSERT_TAIL(head, elm, field)
```

定义

链表

```
// include/queue.h

#ifndef _SYS_QUEUE_H_
#define _SYS_QUEUE_H_
```

```
/*
这个文件定义了三种数据结构类型：列表、尾队列和循环队列。
一个列表以一个单向指针（或一个指向哈希表头的单向指针数组）为首。这些元素是双向链接的，因此可以在不需要遍历列表的情况下删除任意元素。新元素可以在现有元素之前或之后添加到列表中，也可以添加到列表的开头。列表只能按正向方向遍历。
一个尾队列以一对指针为首，一个指向列表的头部，另一个指向列表的尾部。这些元素是双向链接的，因此可以在不需要遍历列表的情况下删除任意元素。新元素可以在现有元素之前或之后添加到列表中，也可以添加到列表的开头或结尾。尾队列只能按正向方向遍历。
一个循环队列以一对指针为首，一个指向列表的头部，另一个指向列表的尾部。这些元素是双向链接的，因此可以在不需要遍历列表的情况下删除任意元素。新元素可以在现有元素之前或之后添加到列表中，也可以添加到列表的开头或结尾。循环队列可以向前或向后遍历，但是其结束列表检测比较复杂。
有关这些宏的使用细节，请参阅queue(3)手册页。
*/

/*
 * List declarations.
 */

/*
一个列表由由LIST_HEAD宏定义的结构体作为其首部。这个结构体包含一个指向列表第一个元素的指针。元素是双向链接的，因此可以在不需要遍历列表的情况下删除任意元素。新元素可以添加到列表中一个现有元素之后或者添加到列表的头部。一个LIST_HEAD结构体可以声明如下：

  LIST_HEAD(HEADNAME, TYPE) head;

其中，HEADNAME是要定义的结构体的名称，TYPE是要链接到列表中的元素类型。
*/
#define LIST_HEAD(name, type)
                \
        struct name {
                \
                 struct type *lh_first; /* first element */
               \
        }

/*
 * Set a list head variable to LIST_HEAD_INITIALIZER(head)
 * to reset it to the empty list.
 */
#define LIST_HEAD_INITIALIZER(head)
                \
        { NULL }

/*
 * Use this inside a structure "LIST_ENTRY(type) field" to use
 * x as the list piece.
 *
 * The le_prev points at the pointer to the structure containing
 * this very LIST_ENTRY, so that if we want to remove this list entry,
 * we can do *le_prev = le_next to update the structure pointing at us.
 */
```

```
#define LIST_ENTRY(type)                                                \
        struct {                                                        \
                struct type *le_next;  /* next element */               \
                struct type **le_prev; /* address of previous next element */ \
        }

/*
 * List functions.
 */

/*
 * Detect the list named "head" is empty.
 */
#define LIST_EMPTY(head) ((head)->lh_first == NULL)

/*
 * Return the first element in the list named "head".
 */
#define LIST_FIRST(head) ((head)->lh_first)

/*
 * Iterate over the elements in the list named "head".
 * During the loop, assign the list elements to the variable "var"
 * and use the LIST_ENTRY structure member "field" as the link field.
 */
#define LIST_FOREACH(var, head, field)                                  \
        for ((var) = LIST_FIRST((head)); (var); (var) = LIST_NEXT((var), field))

/*
 * Reset the list named "head" to the empty list.
 */
#define LIST_INIT(head)                                                 \
        do {                                                            \
                LIST_FIRST((head)) = NULL;                              \
        } while (0)

/*
 * Insert the element 'elm' *after* 'listelm' which is already in the list. The
'field'
 * name is the link element as above.
 *
 * Hint:
 * Step 1: assign 'elm.next' from 'listelm.next'.
```

```
 * Step 2: if 'listelm.next' is not NULL, then assign 'listelm.next.pre' from a
proper value.
 * Step 3: assign 'listelm.next' from a proper value.
 * Step 4: assign 'elm.pre' from a proper value.
 */
#define LIST_INSERT_AFTER(listelm, elm, field)
              \
        /* Exercise 2.2: Your code here. */  \
        do { \
                LIST_NEXT((elm), field) = LIST_NEXT((listelm), field); \
                if (LIST_NEXT((listelm), field) != NULL){ \
                        LIST_NEXT((listelm), field)->field.le_prev =
&LIST_NEXT((elm), field);} \
                LIST_NEXT((listelm), field) = (elm); \
                (elm)->field.le_prev = &LIST_NEXT((listelm), field); \
        } while(0)
/*
 * Insert the element "elm" *before* the element "listelm" which is
 * already in the list.  The "field" name is the link element
 * as above.
 */
#define LIST_INSERT_BEFORE(listelm, elm, field)
               \
        do {
              \
                (elm)->field.le_prev = (listelm)->field.le_prev;
              \
                LIST_NEXT((elm), field) = (listelm);
              \
                *(listelm)->field.le_prev = (elm);
              \
                (listelm)->field.le_prev = &LIST_NEXT((elm), field);
              \
        } while (0)

/*
 * Insert the element "elm" at the head of the list named "head".
 * The "field" name is the link element as above.
 */
#define LIST_INSERT_HEAD(head, elm, field)
              \
        do {
              \
                if ((LIST_NEXT((elm), field) = LIST_FIRST((head))) != NULL)
               \
                        LIST_FIRST((head))->field.le_prev = &LIST_NEXT((elm),
field);              \
                LIST_FIRST((head)) = (elm);
               \
                (elm)->field.le_prev = &LIST_FIRST((head));
               \
        } while (0)
```

```
#define LIST_NEXT(elm, field) ((elm)->field.le_next)

/*
 * Remove the element "elm" from the list.
 * The "field" name is the link element as above.
 */
#define LIST_REMOVE(elm, field)
                \
        do {
                \
                if (LIST_NEXT((elm), field) != NULL)
                \
                        LIST_NEXT((elm), field)->field.le_prev = (elm)-
>field.le_prev;             \
                *(elm)->field.le_prev = LIST_NEXT((elm), field);
                \
        } while (0)

/*
 * Tail queue definitions.
 */
#define _TAILQ_HEAD(name, type, qual)
                \
        struct name {
                \
                qual type *tqh_first;      /* first element */
                \
                qual type *qual *tqh_last; /* addr of last next element */
                \
        }
#define TAILQ_HEAD(name, type) _TAILQ_HEAD(name, struct type, )

#define TAILQ_HEAD_INITIALIZER(head)
                \
        { NULL, &(head).tqh_first }

#define _TAILQ_ENTRY(type, qual)
                \
        struct {
                \
                qual type *tqe_next;       /* next element */
                \
                qual type *qual *tqe_prev; /* address of previous next element */
                \
        }
#define TAILQ_ENTRY(type) _TAILQ_ENTRY(struct type, )
```

尾队列

```
// include/queue.h
#define TAILQ_INIT(head)
                \
        do {
                \
                  (head)->tqh_first = NULL;
                 \
                  (head)->tqh_last = &(head)->tqh_first;
                \
        } while (/*CONSTCOND*/ 0)

#define TAILQ_INSERT_HEAD(head, elm, field)
                 \
        do {
                \
                  if (((elm)->field.tqe_next = (head)->tqh_first) != NULL)
                \
                          (head)->tqh_first->field.tqe_prev = &(elm)->field.tqe_next;
                 \
                  else
                \
                          (head)->tqh_last = &(elm)->field.tqe_next;
                \
                  (head)->tqh_first = (elm);
                \
                  (elm)->field.tqe_prev = &(head)->tqh_first;
                 \
        } while (/*CONSTCOND*/ 0)

#define TAILQ_INSERT_TAIL(head, elm, field)
                 \
        do {
                \
                  (elm)->field.tqe_next = NULL;
                 \
                  (elm)->field.tqe_prev = (head)->tqh_last;
                \
                  *(head)->tqh_last = (elm);
                \
                  (head)->tqh_last = &(elm)->field.tqe_next;
                \
        } while (/*CONSTCOND*/ 0)

#define TAILQ_INSERT_AFTER(head, listelm, elm, field)
                 \
        do {
                \
                  if (((elm)->field.tqe_next = (listelm)->field.tqe_next) != NULL)
                \
```

```
                        (elm)->field.tqe_next->field.tqe_prev = &(elm)-
>field.tqe_next;            \
                else
                \
                        (head)->tqh_last = &(elm)->field.tqe_next;
                \
                (listelm)->field.tqe_next = (elm);
                \
                (elm)->field.tqe_prev = &(listelm)->field.tqe_next;
                \
        } while (/*CONSTCOND*/ 0)

#define TAILQ_INSERT_BEFORE(listelm, elm, field)
                \
        do {
                \
                (elm)->field.tqe_prev = (listelm)->field.tqe_prev;
                \
                (elm)->field.tqe_next = (listelm);
                \
                *(listelm)->field.tqe_prev = (elm);
                \
                (listelm)->field.tqe_prev = &(elm)->field.tqe_next;
                \
        } while (/*CONSTCOND*/ 0)

#define TAILQ_REMOVE(head, elm, field)
                \
        do {
                \
                if (((elm)->field.tqe_next) != NULL)
                \
                        (elm)->field.tqe_next->field.tqe_prev = (elm)-
>field.tqe_prev;             \
                else
                \
                        (head)->tqh_last = (elm)->field.tqe_prev;
                \
                *(elm)->field.tqe_prev = (elm)->field.tqe_next;
                \
        } while (/*CONSTCOND*/ 0)

#define TAILQ_FOREACH(var, head, field)
                \
        for ((var) = ((head)->tqh_first); (var); (var) = ((var)->field.tqe_next))

#define TAILQ_FOREACH_REVERSE(var, head, headname, field)
                \
        for ((var) = (*(((struct headname *)((head)->tqh_last))->tqh_last)); (var);
                \
             (var) = (*(((struct headname *)((var)->field.tqe_prev))->tqh_last)))
```

```
#define TAILQ_CONCAT(head1, head2, field)
                \
        do {
                \
                  if (!TAILQ_EMPTY(head2)) {
                \
                          *(head1)->tqh_last = (head2)->tqh_first;
                \
                          (head2)->tqh_first->field.tqe_prev = (head1)->tqh_last;
                 \
                          (head1)->tqh_last = (head2)->tqh_last;
                \
                          TAILQ_INIT((head2));
                \
                  }
                 \
        } while (/*CONSTCOND*/ 0)

/*
 * Tail queue access methods.
 */
#define TAILQ_EMPTY(head) ((head)->tqh_first == NULL)
#define TAILQ_FIRST(head) ((head)->tqh_first)
#define TAILQ_NEXT(elm, field) ((elm)->field.tqe_next)

#define TAILQ_LAST(head, headname) (*(((struct headname *)((head)->tqh_last))-
>tqh_last))
#define TAILQ_PREV(elm, headname, field) (*(((struct headname *)((elm)-
>field.tqe_prev))->tqh_last))

#endif
```