EMBEDDED DEVICE DRIVERS

Linux Device Drivers on Beaglebone Black

Linked Lists

- Linked list is a data structure
  - Sequence of nodes
  - Each connected to the next and/or previous
    - Using pointers
  - Containing some data / variable

- Linked list types:
  - Singly linked list
    - Only next pointer in a node
  - Doubly linked list
    - Next and previous pointers in a node
  - Circular linked list
    - Head connected to tail (and vice versa if doubly linked)

- Linux kernel has its own implementation of linked list
  - Since they are so useful and are needed for a lot of operations

LKM: Kernel linked list (1/5)

- Usually, we declare linked list nodes thus

```
struct my_node {
        int data;
        struct my_node *next; // optionally, struct my_node *prev;
        };
```

- But the kernel implementation is different
  - We start with:

```
struct my_node {
        int data;
        };
```

  - Then we use add the kernel's ***struct list_head*** to 'connect' it to a linked list

```
struct my_node {
        int data;
        struct list_head my_list;
        };
```

- Definition inside the kernel (<linux/list.h>

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

LKM: Kernel linked list (2/5)

- Creating nodes in the kernel LL
  - Option #1

```
struct my_node node1 {
        .data = 10,
        .my_list = LIST_HEAD_INIT(node1.my_list)
        };
```

  - Option #2

```
struct my_node node2;
node2.data = 20;
INIT_LIST_HEAD(&node2.my_list);
```

  - Option #3

```
struct *my_node node3 = kmalloc(sizeof(struct my_node), GFP_KERNEL);
node3->data = 30;
INIT_LIST_HEAD(&node3->my_list);
```

Definition inside the kernel:

```
#define LIST_HEAD_INIT(name)                    { &(name, &(name) }

static inline void INIT_LIST_HEAD(struct list_head *list)
{
                list->next = list; list->prev = list;
}
```

LKM: Kernel linked list (3/5)

- Creating a head node for our linked list
  ***LIST_HEAD**(my_list_head);*

  - Definition inside the kernel

  *#define **LIST_HEAD**(name) *
  *struct list_head name = **LIST_HEAD_INIT**(name)*

- Adding elements (nodes) to the linked list
  - Add after a head – used for stacks
    ***list_add**(&node1.my_list, &my_list_head);*
    ***list_add**(&node2.my_list, &my_list_head);*

  - Add before a head – used for queue
    ***list_add_tail**(&node1.my_list, &my_list_head);*
    ***list_add_tail**(&node2.my_list, &my_list_head);*

LKM: Kernel linked list (4/5)

- Node deletion
  - *inline void **list_del**(struct list_head *entry);*
- Node replacement
  - *inline void **list_replace**(struct list_head *old, struct list_head *new);*
- Node movement
  - *inline void **list_move**(struct list_head *entry, struct list_head *head);*
    - Move to after the head
  - *inline void **list_move_tail**(struct list_head *entry, struct list_head *head);*
    - Move to before the head
- Checks

  *inline int **list_is_last**(const struct list_head *entry, const struct list_head *head);*
  *inline int **list_empty**(const struct list_head *head);*
  *inline int **list_is_singular**(const struct list_head *head);*

LKM: Kernel linked list (5/5)

- Forward traversal

```
struct my_node *node;
list_for_each_entry ( node, &my_list_head, my_list )
{   pr_info("%d\n", node->data;    }
```

- Reverse traversal

```
struct my_node *node;
list_for_each_entry_reverse( node, &my_list_head, my_list )
{   pr_info("%d\n", node->data   }
```

- Deleting the linked list by traversal

```
struct my_node *node, *tmp;
list_for_each_entry_safe( node, tmp, my_list_head, my_list )
{
  list_del ( &node->my_list );
  kfree(node); // if node was kmalloc'd
}
```

LKM: Why use kernel LL

- Advantages of this approach:
  - Any kind of data can be ‘connected’ to a linked list
  - Same data can be part of multiple linked lists

- Why use kernel LL
  - Avoids reinventing the wheel
  - Avoids duplication of code and efforts
  - Well-tested and stable code base
    - Used by lots of modules and kernel code

LKM: Kernel LL Exercise

- Refer ***mod9*** directory
  - ***mod91.c*** contains code for linked list
    - With statically allocated nodes
      - 2 options are exercised
    - The module_init() function creates and prints the linked list
    - The module_exit() function deletes the linked list
  - ***mod92.c*** contains code for linked list
    - With dynamically allocated nodes
    - The module_init() creates and prints LL forwards and backwards
    - The module_exit() deletes the list via traversal
  - ***mod93.c*** contains code for list node replacement
    - With dynamically allocated nodes
    - In module_init(), we replace node with data=50 by 1000
      - And confirm it by printing
    - module_exit() deletes the list via traversal

THANK YOU!