Linux kernel data structure

XArray

Practical Class 8

Systems and Storage Laboratory
Department of Computer Science and Engineering
Chung-Ang University

Index

Intodocing XArray

- **eXtensible Arrays**
  - Abstract data type which behaves like a very large array of pointers
  - Normal pointers may be stored in the XArray directly
  - They must be 4-byte aligned, which is true for any pointer returned from `kmalloc()` and `alloc_page()`
  - Meets many of the same needs as a hash or a conventional resizable array
  - More memory-efficient, parallelisable and cache friendly than a doubly-linked list
  - Uses RCU and an internal spinlock to synchronize access
  - The most important user of the XArray is the page cache

XArray structure

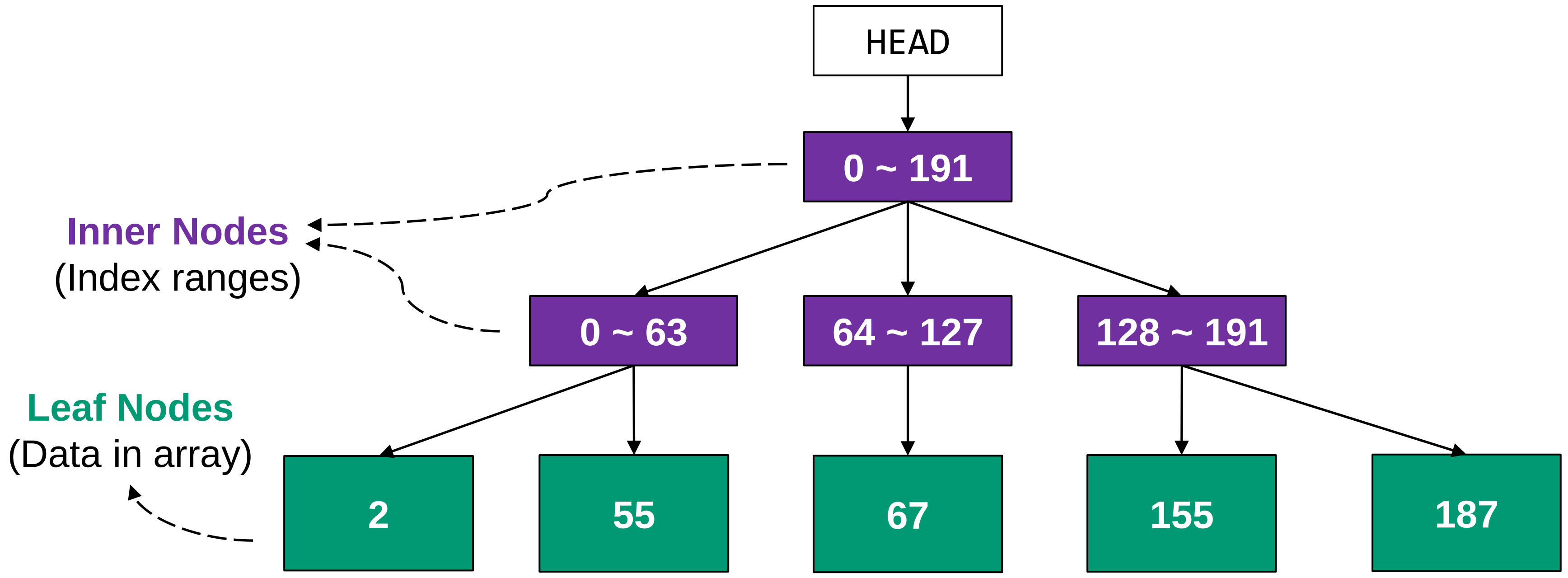
- **The anchor of the XArray**
  - It is a very small data structure so it could be easily defined statically or embed

```
struct xarray {
    spinlock_t  xa_lock;
    gfp_t       xa_flags;
    void __rcu *    xa_head;
};
```

  - **xa_lock**: could be used to protect the contents of XArray
  - **xa_head**: the root node of XArray

XArray node

- The data structure of XArray nodes is **`struct xa_node`**
  - The data is only kept in leaf nodes
  - Inner nodes save index ranges
  - Inner nodes are only for navigating



struct xa_node

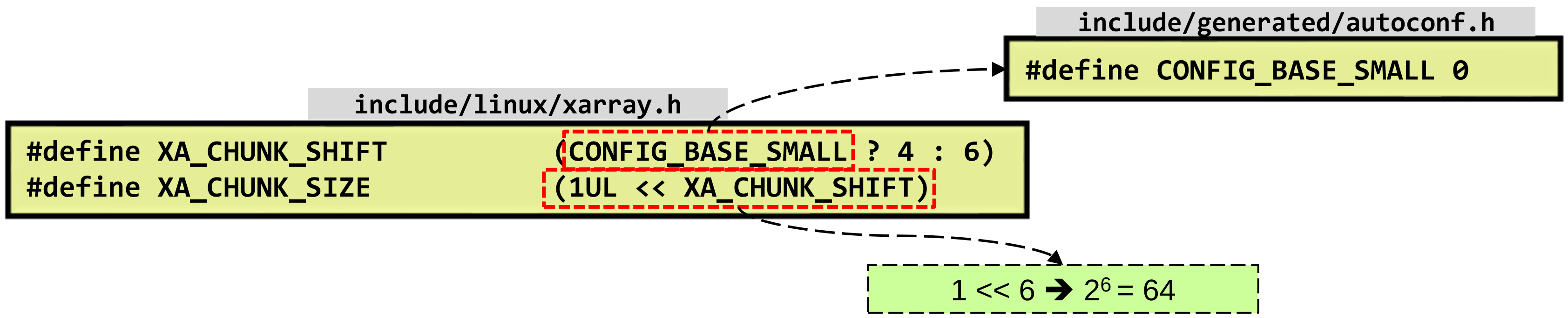
- An element in XArray

```
struct xa_node {
    unsigned char   shift;       /* Bits remaining in each slot */
    unsigned char   offset;      /* Slot offset in parent */
    unsigned char   count;       /* Total entry count */
    unsigned char   nr_values;   /* Value entry count */
    struct xa_node __rcu *parent;    /* NULL at top of tree */
    struct xarray   *array;      /* The array we belong to */
    union {
        struct list_head private_list;   /* For tree user */
        struct rcu_head rcu_head;    /* Used when freeing node */
    };
    void __rcu  *slots[XA_CHUNK_SIZE];
    union {
        unsigned long   tags[XA_MAX_MARKS][XA_MARK_LONGS];
        unsigned long   marks[XA_MAX_MARKS][XA_MARK_LONGS];
    };
};
```

CAU SySLAB Systems and Storage Laboratory

struct xa_node

- **shift:** bits remaining in each slot (4 or 6)
- **offset:** the slot offset in **parent**
- **count:** the count of every non-NULL element in the slots array
- **nr_values:** the count of every element in slots which is either a value entry or a sibling of a value entry
- **array:** the xarray that the nodes belong to
- **slots:** an array saving children nodes with `XA_CHUNK_SIZE` elements
  - Default of `XA_CHUNK_SIZE` is 64 – with `XA_CHUNK_SHIFT` = 6



include/linux/xarray.h

```
#define XA_CHUNK_SHIFT          (CONFIG_BASE_SMALL ? 4 : 6)
#define XA_CHUNK_SIZE           (1UL << XA_CHUNK_SHIFT)
```

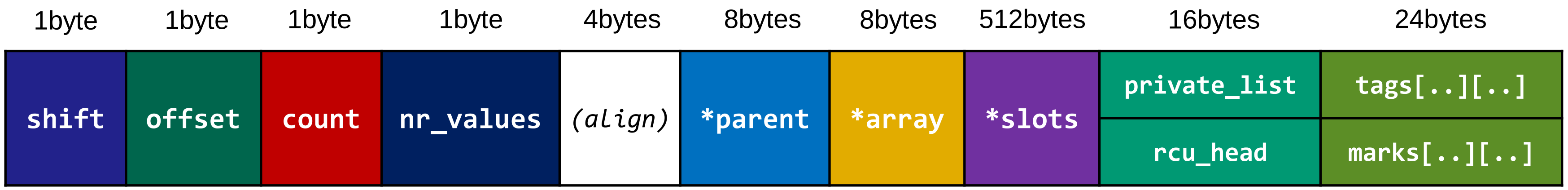
include/generated/autoconf.h

```
#define CONFIG_BASE_SMALL 0
```

$1 << 6 \rightarrow 2^6 = 64$

struct xa_node

- **64bit Architecture**
  - 576bytes per one xa_node structure
  - 4KB page can have up to 7 nodes

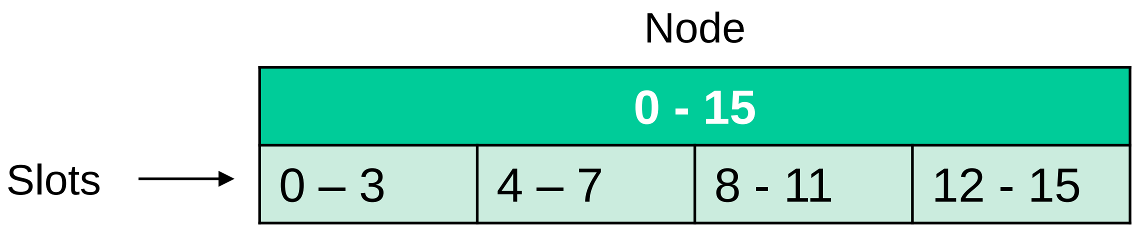


struct xa_node

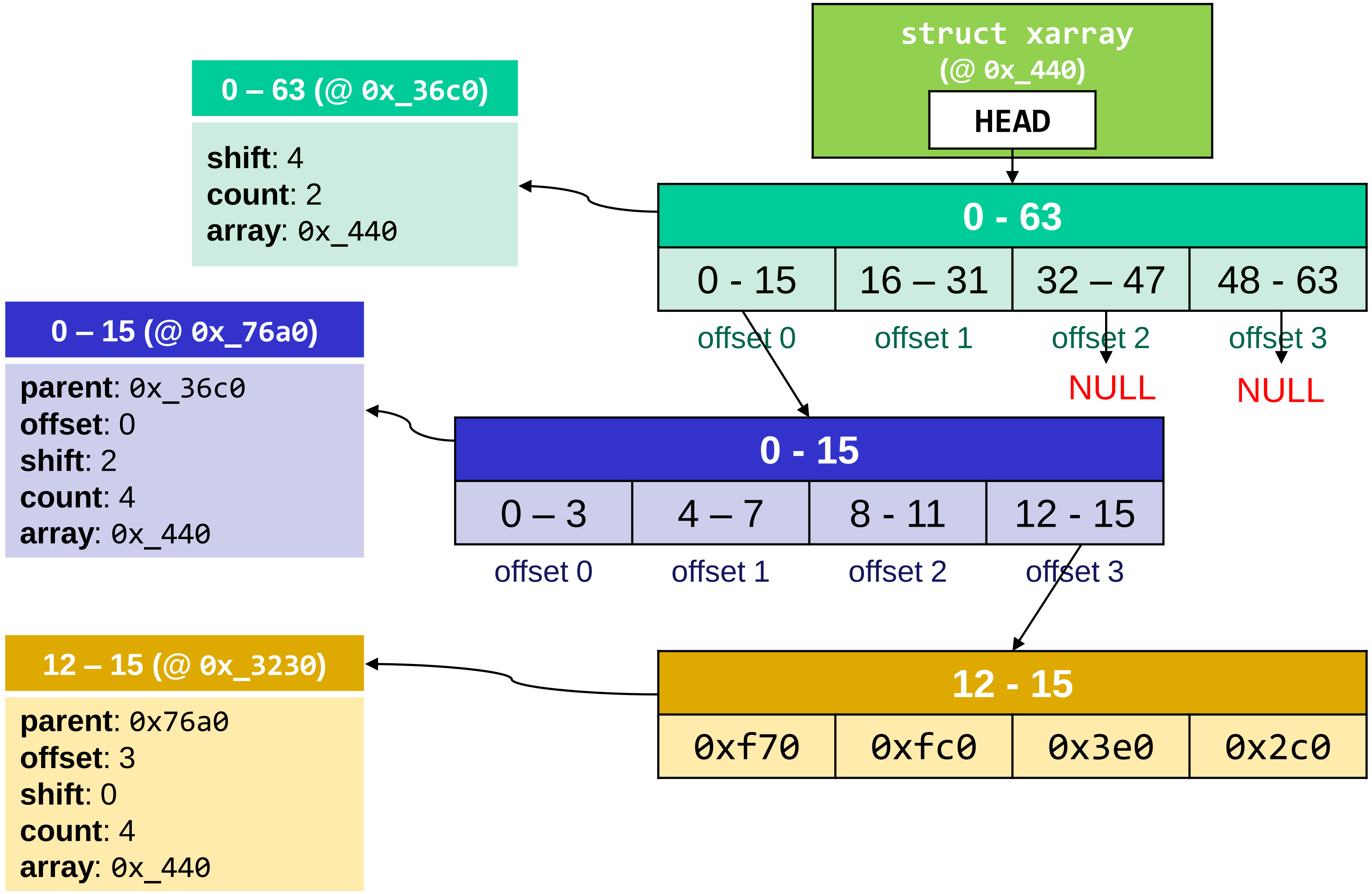
XArray node slots

- **Slots can contain inner nodes or leaf nodes**
  - Ex) Let `XA_CHUNK_SIZE` = 4
    - One node has 4 slots
    - `XA_CHUNK_SHIFT` = 2

```
#define XA_CHUNK_SIZE          (1UL << XA_CHUNK_SHIFT)
```

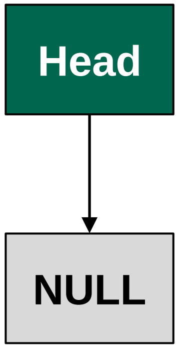


XArray node slots



XArray node insertion routines

❖ Initializing Xarray



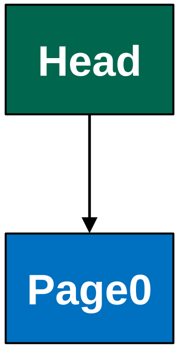
```
struct xarray {
    spinlock_t  xa_lock;
    gfp_t       xa_flags;
    void __rcu *    xa_head;
};
```

Xarray structure	
xa_head	NULL
xa_lock	initialized
xa_head	NULL

Xarray node structure fields	
shift	0
offset	0
count	0
nr_values	0
parent	NULL
xarray	NULL

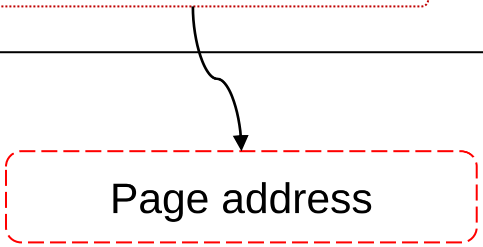
XArray node insertion routines

- **Insert first element**



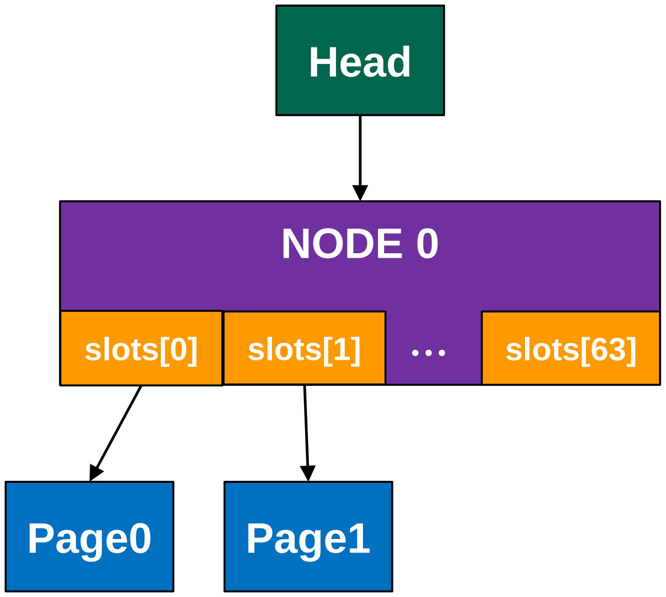
Xarray structure	
xa_head	NULL
xa_lock	initialized
xa_head	page0 address

```
struct xarray {
    spinlock_t  xa_lock;
    gfp_t       xa_flags;
    void __rcu *     xa_head;
};
```



XArray node insertion routines

- **Insert second element**

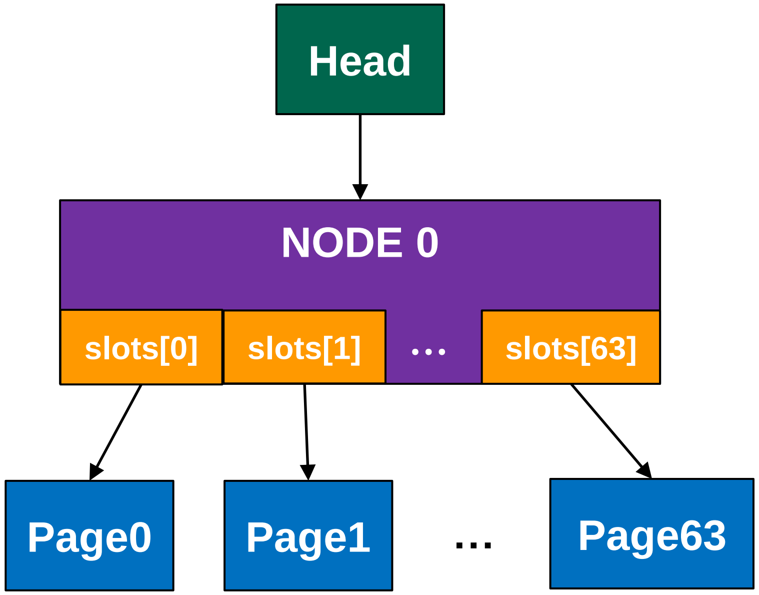


Xarray structure	
`xa_head`	`NULL`
`xa_lock`	`initialized`
`xa_head`	`node0 address`

Xarray node structure fields		
shift	0	
offset	0	
count	2	
nr_values	0	
parent	head	
array	0x01	
slots	[0]	P0 addr
	[1]	P1 addr

XArray node insertion routines

- Insert 64th element

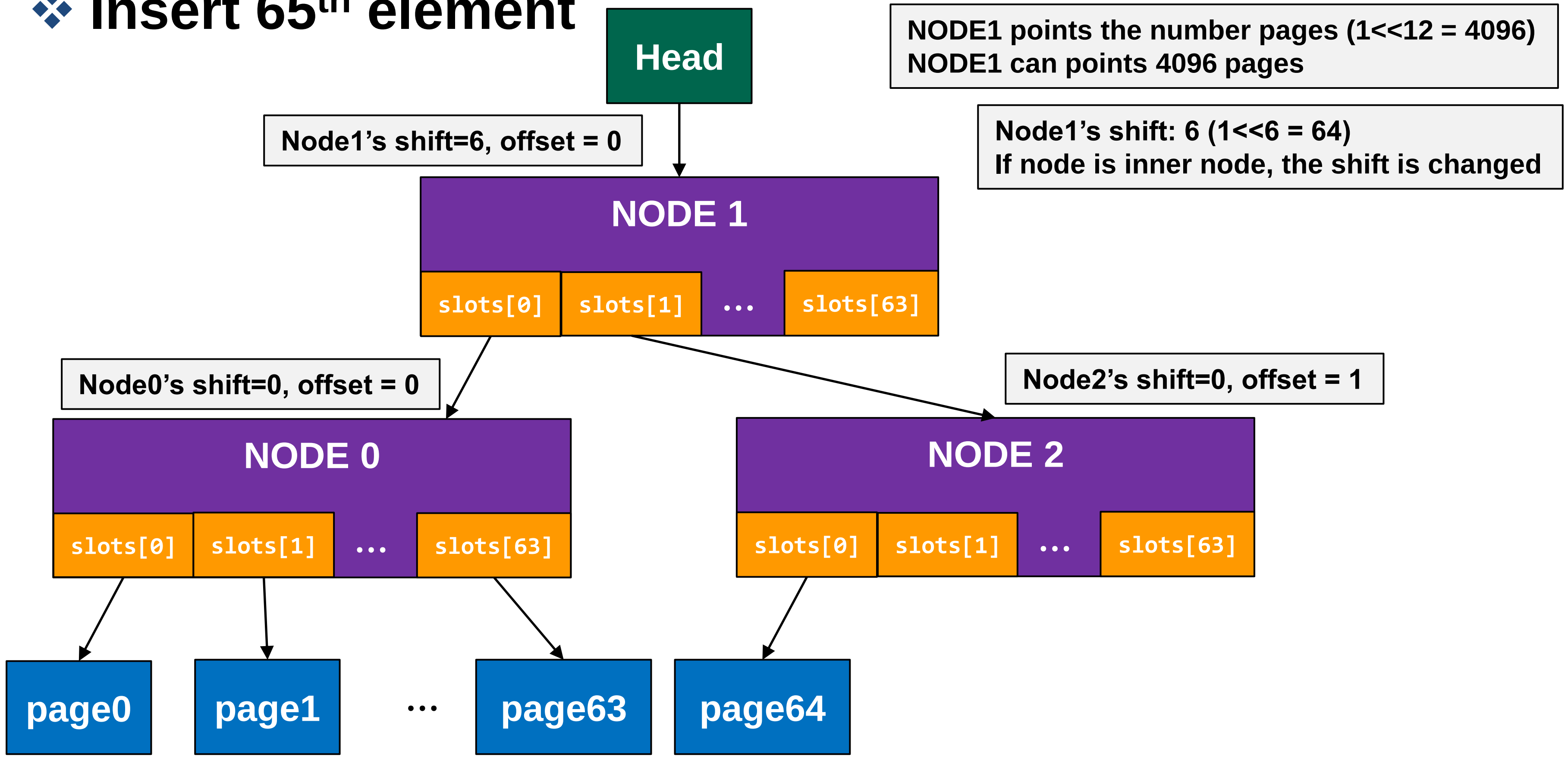


Xarray structure	
xa_head	NULL
xa_lock	initialized
xa_head	node0 address

Xarray node structure fields	
shift	0
offset	0
count	**64**
nr_values	0
parent	head
array	0x01
slots[0]~[63]	**P0 addr ~ P63 addr**

XArray node insertion routines

- Insert 65th element


Head
NODE1 points the number pages (1<<12 = 4096)
NODE1 can points 4096 pages
Node1's shift=6, offset = 0
Node1's shift: 6 (1<<6 = 64)
If node is inner node, the shift is changed
NODE 1
slots[0]
slots[1]
…
slots[63]
Node0's shift=0, offset = 0
Node2's shift=0, offset = 1
NODE 0
slots[0]
slots[1]
…
slots[63]
NODE 2
slots[0]
slots[1]
…
slots[63]
page0
page1
…
page63
page64

```
Xarray structure: xa_head = NULL; xa_lock is initialized; xa_head = page address
Xarray node structure 0 fields: shift=0, offset=0, count=64, nr_values=0, parent=node1, xarray = 0x01,
                                slots[0] ~ slots[63] = P0 addr ~ P63 addr
Xarray node structure 1 fields: shift=6, offset=0, count=2, nr_values=0, parent=head, xarray = 0x01,
                                slots[0] = node0, slots[1] = node2
Xarray node structure 2 fields: shift=0, offset=1, count=1, nr_values=0, parent=node1, xarray = 0x01,
                                slots[0] = P64 addr
```

XArray Normal APIs

Search

- Search the XArray for an entry

```
void *xa_find(struct xarray *xa, unsigned long *indexp,
              unsigned long max, xa_mark_t filter)
```

  - **xa**: XArray to search entry from
  - **indexp**: Pointer to an index. (Found entry index will be saved here)
  - **max**: Maximum index to search to.
  - **filter**: Selection criterion.
  - Return: The entry, if found, otherwise ***`NULL`***

- Load an entry from an XArray

```
void *xa_load(struct xarray *xa, unsigned long index);
```

  - **xa**: XArray to load entry from
  - **index**: Entry index that we are searching for
  - Return: The entry at ***`index`*** in ***`xa`***.

XArray Normal APIs

Insert

- Store this entry in the XArray unless another entry is already present

```
int xa_insert(struct xarray *xa, unsigned long index,
              void *entry, gfp_t gfp)
```

- **xa**: XArray.
- **index**: Index into array.
- **entry**: New entry.
- **gfp**: Memory allocation flags
- Return:
  - ✓ **`0`** if the store succeeded.
  - ✓ ***`-EBUSY`*** if another entry was present.
  - ✓ ***`-ENOMEM`*** if memory could not be allocated.

XArray Normal APIs

- **Update**
  - Store this entry in the XArray

```
void *xa_store(struct xarray *xa, unsigned long index,
               void *entry, gfp_t gfp)
```

- **xa**: XArray.
- **index**: Index into array to update with.
- **entry**: New entry to store.
- **gfp**: Memory allocation flags.
- Return:
  - ✓ The old entry at this index on success
  - ✓ **`xa_err(-EINVAL)`** if ***`entry`*** cannot be stored in an XArray
  - ✓ **`xa_err(-ENOMEM)`** if memory allocation failed

XArray Normal APIs

- **Delete**
  - Erase this entry from the Xarray

```
void *xa_erase(struct xarray *xa, unsigned long index);
```

    - **xa**: XArray.
    - **index**: Index of entry to delete from ***xa***.
    - Return: The entry which used to be at this index

ds_monitoring

Xarray Usage Example

Purpose of ds_monitoring

- **To monitor values in overall data structure**
  - Functions in the Linux kernel are called too many times
  - There is a limit to checking values only with `printk()`
  - Collecting the data in one place and printing them out all at once later will be one solution
- **XArray will easily handle the problem, according to the goal**
  - XArray is Linux kernel's native data structure
  - It has advantages in terms of stability, reliability and efficiency.

struct ds_monitoring structure

- Main container for the XArray of elements

```
struct ds_monitoring {
        struct xarray *elements;
        unsigned long long total_counts;
        const struct ds_monitoring_operations *dm_ops;
};
```

  - **elements**: pointer to the XArray which contains pointers to each ds_monitoring_elem
  - **total_counts**: counter for how many times have xarray been called
  - **dm_ops**: member functions for manipulating and managing elements

struct ds_monitoring_operations

- **Collection of pointers to functions for manipulating and managing elements**
  - User must implement functions which fit for purpose, and connect them to this structure
  - Each function pointers will be embedded into ds_monitoring's own functions

```
struct ds_monitoring_operations {
        unsigned long (*get_index)(void *elem);
        const char * (*get_name)(void *elem);
        void (*print_elem)(
                                unsigned long index,
                                const char *name,
                                unsigned long long count,
                                int percentage
        );
};
```

struct ds_monitoring_elem

- **Element of XArray**

```
struct ds_monitoring_elem {
        unsigned long key;
        char *name;
        unsigned long long count;
};
```

  - **key**: index of the element itself in XArray
  - **name**: representative name for this element, will be used when printing elements
  - **count**: counter for how many times have this element been searched from XArray

DEFINE_DS_MONITORING

- **Defines a Data Structure Monitoring**
  - This is intended for file scope definitions of Data Structure Monitoring.
  - It declares and initializes an empty Data Structure Monitoring structure with the chosen name
  - It does the initialization at compile time instead of runtime

```
#define DEFINE_DS_MONITORING(name, get_idx_fn, get_name_fn, print_fn)     \
        DEFINE_XARRAY(name##xarray);                                      \
        DEFINE_DS_MONITORING_OPS(name, get_idx_fn, get_name_fn, print_fn);\
        struct ds_monitoring name = DS_MONITORING_INIT(name##xarray, name##_dm_ops);
```

- **name**: A string that names your Data Structure Monitoring
- **get_idx_fn**: Address of function which prepares key for a ds_monitoring_elem
- (Optional) **get_name_fn**: Address of function which determines name of each ds_monitoring_elem (pass `NULL` if not used)
- **print_fn**: A function address which prints out for each element

DEFINE_DS_MONITORING_OPS

- **Defines ds_monitoring_operations and map each function pointers with real function addresses**

```
#define DEFINE_DS_MONITORING_OPS(name, get_idx_fn, get_name_fn, print_fn)  \
        static const struct ds_monitoring_operations name##_dm_ops = {     \
                .get_index = get_idx_fn,                            \
                .get_name = get_name_fn,                            \
                .print_elem = print_fn,                             \
        }
```

- **name**: A string of your ds_monitoring name
- **get_idx_fn**: A function address which prepares key for a ds_monitoring_elem
- **get_name_fn**: A function address which determines name of each ds_monitoring_elem
- **print_fn**: A function address which prints out for each element

DS_MONITORING_INIT

- **Macro for Initializing ds_monitoring**

```
#define DS_MONITORING_INIT(xarray, _dm_ops)    \
 {                                            \
        .elements = &xarray                   \
        .total_counts = 0,                    \
        .dm_ops = &_dm_ops,                   \
}
```

- **xarray**: XArray to map with elements in ds_monitoring
- **_dm_ops**: ds_monitoring_operations to map with dm_ops in ds_monitoring

DECLARE_DS_MONITORING

- Declares ds_monitoring which was previously defined in another file

```
#define DECLARE_DS_MONITORING (name)          \
        extern struct ds_monitoring name;
```

find_ds_monitoring()

- **Find if ds_monitoring_elem of elem is in Xarray**
  - If found, increase referenced count
  - If not, create a new ds_monitoring_elem for elem

```
void find_ds_monitoring(struct ds_monitoring *dm, void *elem)
{
    struct ds_monitoring_elem *cur;
    unsigned long xa_index;

    if (dm->dm_ops->get_index) {
        xa_index = dm->dm_ops->get_index(elem);

        // search ds_monitoring_elem at index of xa_index from xarray root
        cur = (struct ds_monitoring_elem *) xa_load(dm->elements, xa_index);

        if (cur) {
            __sync_fetch_and_add(&cur->count, 1);
        } else {
            insert_ds_monitoring(dm, xa_index, elem);
        }
        __sync_fetch_and_add(&dm->total_counts, 1);
    }
}
```

insert_ds_monitoring()

- Creates new xa_mapping_elem from elem and append it into the XArray

```
static void
insert_ds_monitoring(struct ds_monitoring *dm, unsigned long index, void *elem)
{
    char *name;
    struct ds_monitoring_elem *new = kmalloc(sizeof(struct ds_monitoring_elem),
                                             GFP_KERNEL);
    new->key = index;
    new->count = 1;

    if (dm->dm_ops->get_name) {
        name = dm->dm_ops->get_name(elem);
        new->name = kmalloc(strlen(name)+1, GFP_KERNEL);
        strcpy(new->name, name);
    } else {
        new->name = NULL;
    }
    xa_store(dm->elements, new->key, (void*) new, GFP_KERNEL);
}
```

print_ds_monitoring()

- Dump all the ds_monitoring_elem information in the ds_monitoring

```
void print_ds_monitoring(struct ds_monitoring* dm)
{
        unsigned long cur_idx;
        void *cur;
        char *cur_name;
        unsigned long long cur_count;
        int percentage;

        if (!dm->total_counts)
                return;

        xa_for_each(dm->elements, cur_idx, cur) {
                cur_name = ((struct ds_monitoring_elem *)cur)->name;
                cur_count = ((struct ds_monitoring_elem *)cur)->count;
                percentage = cur_count  * 100 / dm->total_counts;
                dm->dm_ops->print_elem(cur_idx, cur_name, cur_count, percentage);
        }
}
```

SySLAB Systems and Storage Laboratory

delete_ds_monitoring()

- **Frees all the allocated memory for ds_monitoring, and then destroys it**

```c
void delete_ds_monitoring(struct ds_monitoring *dm)
{
	unsigned long cur_idx;
	void *cur;
	char *cur_name;

	 xa_for_each(dm->elements, cur_idx, cur) {
		cur_name = ((struct ds_monitoring_elem *)cur)->name;
		kfree(cur_name);
		kfree(cur);
	}
	xa_destroy(dm->elements);
}
```

Usage Example: Definition

- **Define your own ds_monitoring at the global scope of the c file you want to use in**
  - In this case, `struct ds_monitoring thread_dm` will be created

```
DEFINE_DS_MONITORING(thread_dm, get_thread_idx, get_thread_name, print_zone_dm);
```

Usage Example: get_idx_fn

- **get_idx_fn should**
  - Receive void* of any structure that you want to watch
  - Return unsigned long value of index of which element should become

```
static unsigned long get_thread_idx(void *elem)
{
        int node_idx;
        unsigned long xa_index;
        int zone_idx;

        struct task_struct *current_task = (struct task_struct *) elem;

        return (unsigned long) current_task->pid;
}
```

Usage Example: get_name_fn

- **get_name_fn should**
  - Receive void* of any structure that you want to watch
  - Return const char* value of name by which element should be called

```
static const char * get_thread_name(void *elem)
{
        struct task_struct *current_task = (struct task_struct *) elem;
        return current_task->comm;
}
```

Systems and Storage Laboratory

Usage Example: print_fn

- **print_fn should receive**
  - unsigned long **index**: key value for each element
  - const char ***name**: representative name for this element
  - unsigned long long **count**: element has been searched
  - int **percentage**: proportion of this element count in the total_counts

```
static void print_zone_dm(unsigned long pid,
        const char *name,
        unsigned long long count,
        int percentage)
{
    printk("thread %s: pid %ld called wb_check_background_flush()  \
        %lld times (%d%%)\n", name, pid, count, percentage);
}
```

Usage Example: find_ds_monitoring

- User will try to find some element in the ds_monitoring

```
#include "ds_monitoring.h"

DEFINE_DS_MONITORING(thread_dm, get_thread_idx, get_thread_name, print_zone_dm);

static long wb_check_background_flush(struct bdi_writeback *wb)
{
    if (wb_over_bg_thresh(wb)) {
        struct wb_writeback_work work = {
            .nr_pages    = LONG_MAX,
            .sync_mode   = WB_SYNC_NONE,
            .for_background = 1,
            .range_cyclic   = 1,
            .reason      = WB_REASON_BACKGROUND,
        };

        find_ds_monitoring(&thread_dm, current);

        return wb_writeback(wb, &work);
    }
    return 0;
}
```

SySLAB Systems and Storage Laboratory

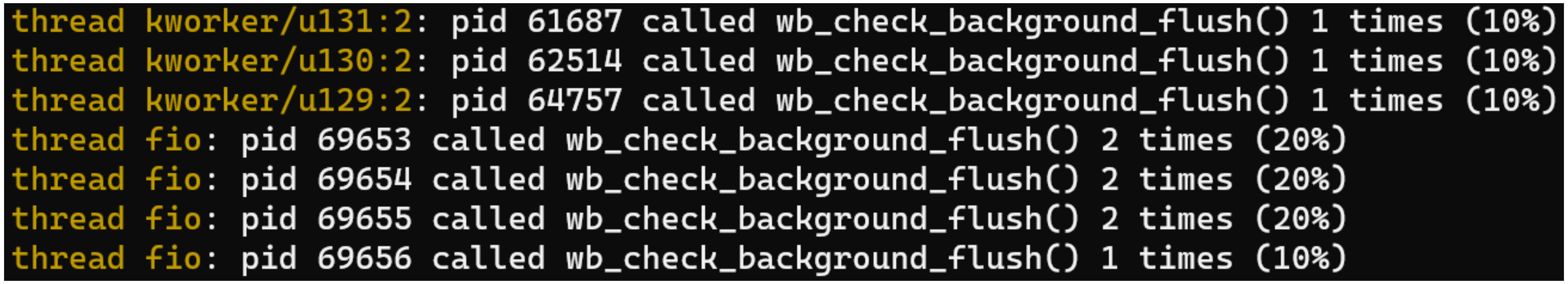
Usage Example: print dm & delete dm

- **Later at some point, you can print out the elements in the ds_monitoring**
- **After finishing utilizing ds_monitoring, make sure you destroy it**

```
#include "ds_monitoring.h"

DECLARE_DS_MONITORING(thread_dm);

static void __exit ext4_exit_fs(void)
{
    ...

    print_ds_monitoring(&thread_dm);

    delete_ds_monitoring(&thread_dm);
}
```

SySLAB Systems and Storage Laboratory

Usage Example: Dump result



```
thread kworker/u131:2: pid 61687 called wb_check_background_flush() 1 times (10%)
thread kworker/u130:2: pid 62514 called wb_check_background_flush() 1 times (10%)
thread kworker/u129:2: pid 64757 called wb_check_background_flush() 1 times (10%)
thread fio: pid 69653 called wb_check_background_flush() 2 times (20%)
thread fio: pid 69654 called wb_check_background_flush() 2 times (20%)
thread fio: pid 69655 called wb_check_background_flush() 2 times (20%)
thread fio: pid 69656 called wb_check_background_flush() 1 times (10%)
```

Overall Form

❖ Monitoring part

```
#include "ds_monitoring.h"

static unsigned long get_target_idx(void *elem)
{
    ...
}

static const char * get_target_name(void *elem)
{
    ...
}

static void print_target_dm(unsigned long index, const char *name,
            unsigned long long count, int percentage)
{
    ...
}

DEFINE_DS_MONITORING(your_dm, get_target_idx, get_target_name, print_target_dm);

int function(...)
{
    struct data_structure target_to_monitor = {
        .foo   = LONG_MAX,
        .bar  = “Hello",
    };
    ...
    find_ds_monitoring(&your_dm, &target_to_monitor);
    ...
    return 0;
}
```

Overall Form

❖ **Dumping result and destroying part**

```
#include "ds_monitoring.h"

DECLARE_DS_MONITORING(your_dm);

static void __exit module_exit_fs(void)
{
      ...

    print_ds_monitoring(&your_dm);

    delete_ds_monitoring(&your_dm);
}
```