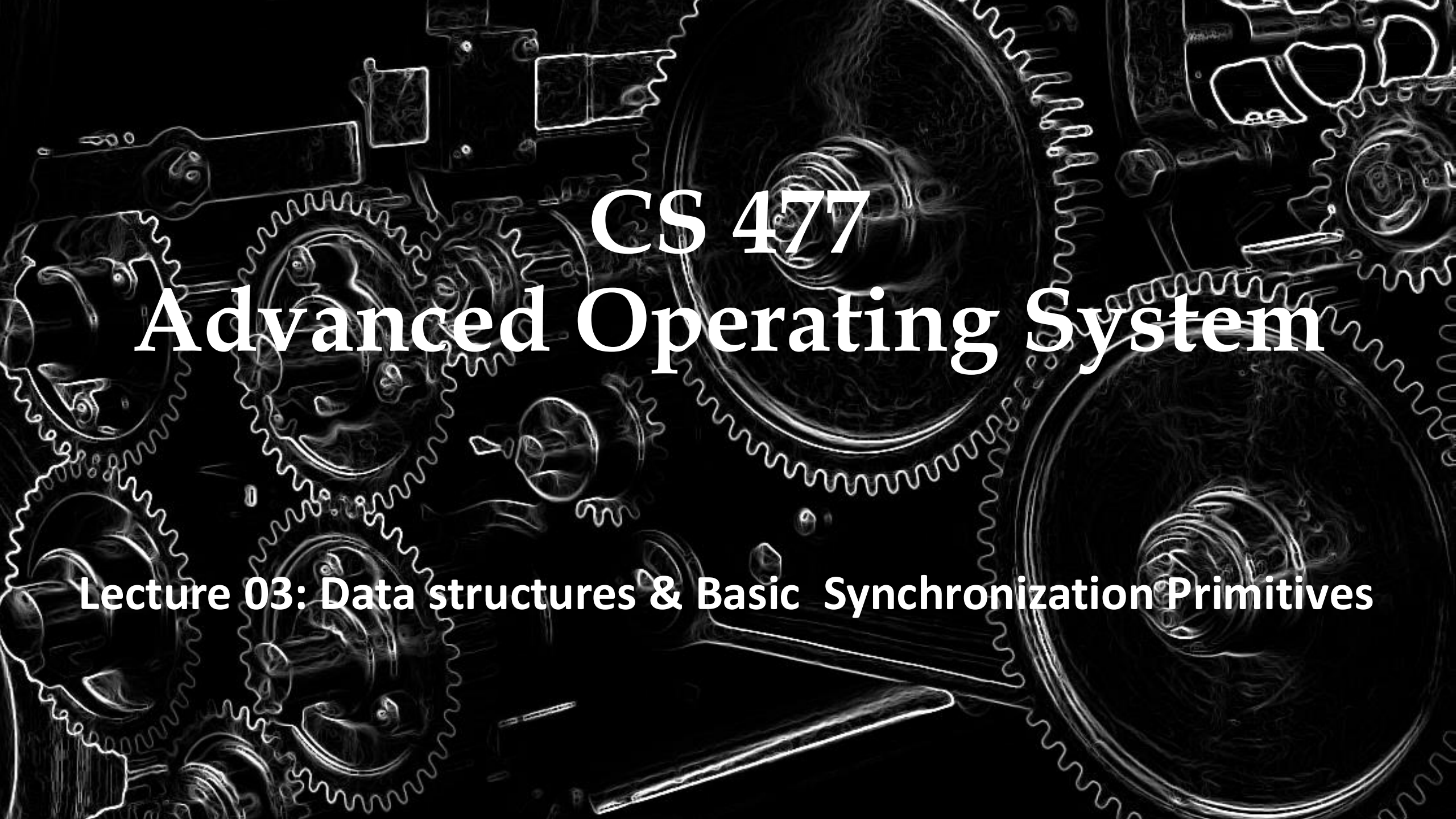


CS 477
Advanced Operating System

Lecture 03: Data structures & Basic Synchronization Primitives

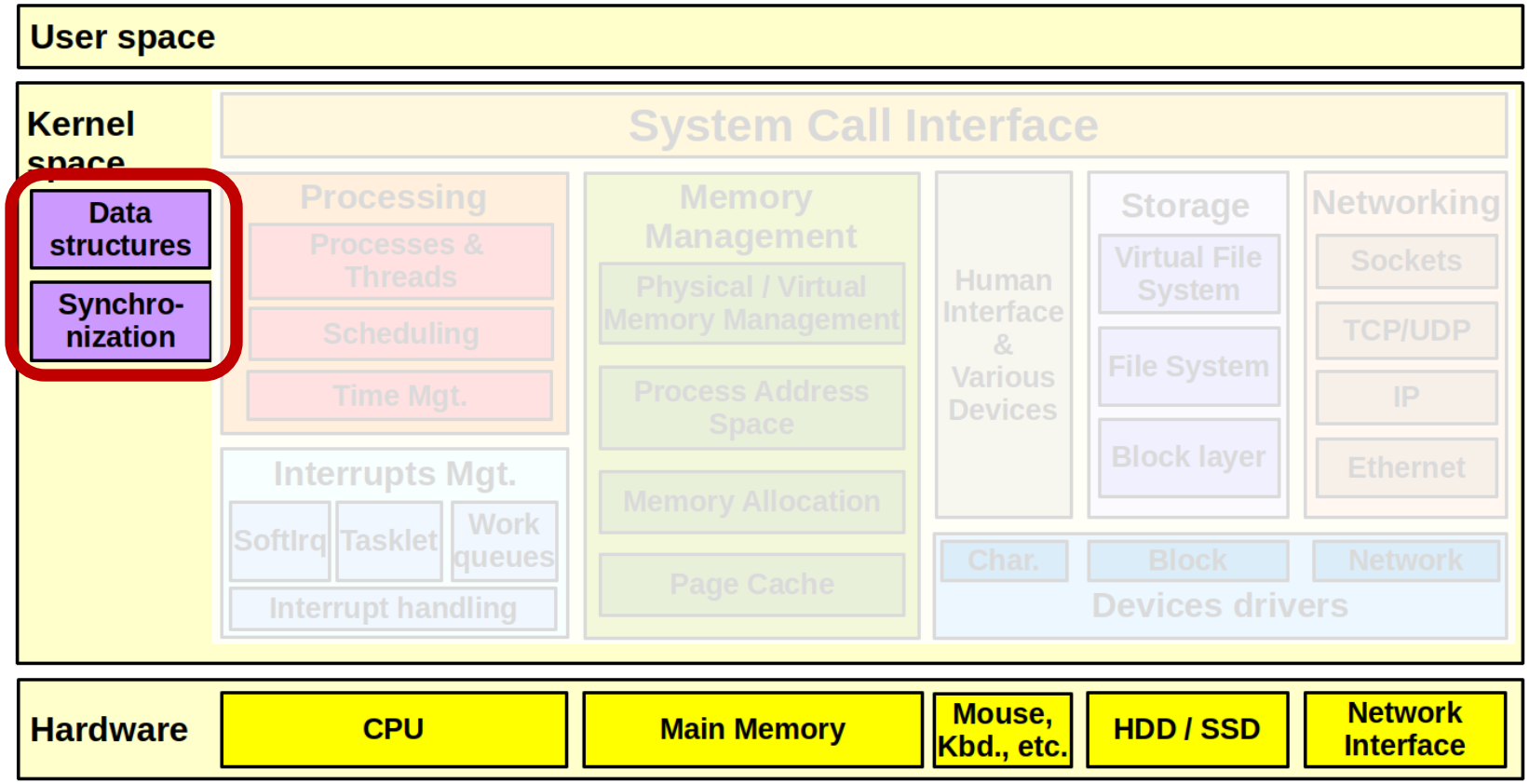
Some updates

- Lab 1 is due this Friday
  - **3** have submitted
- Released project information
  - Form teams and meet with us to scope the project in 3 weeks
- Workload
  - 10-12 hours per week is expected for this course
  - Lab 1 will take a while, but then other labs will not require time-consuming kernel compilation
- Any other suggestions?

Focus of today's lecture

- Kernel data structures
  - Linked list
  - Hash table
  - Red-black tree
  - Xarray
  - Maple tree
- Handling concurrency
  - Atomics
  - Spinlock and its readers-writer variant
  - Mutex and its readers-writer variant

The building blocks used in every kernel subsystem



Why data structures are important?

"I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code and his data structures more important. Bad programmers worry about the code, Good programmers worry about data structures and their relationships."

-- Linus Torvalds

"It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures."

--Alan J. Perlis

Common design patterns for kernel data structures

- **Embed the pointer structure**

  `list_head`, **`hlist_node`**, **`rb_node`**

  - Full control to place fields in the structure
    - Allows placing critical fields closer together → improve cache utilization
  - An object can be put on two or more data structures **independently**
    - Eg, have multiple **`list_head`** fields for putting an object in two lists
  - **`container_of`**, **`list_entry`**, and **`rb_entry`** are used to get their corresponding embedded data structure

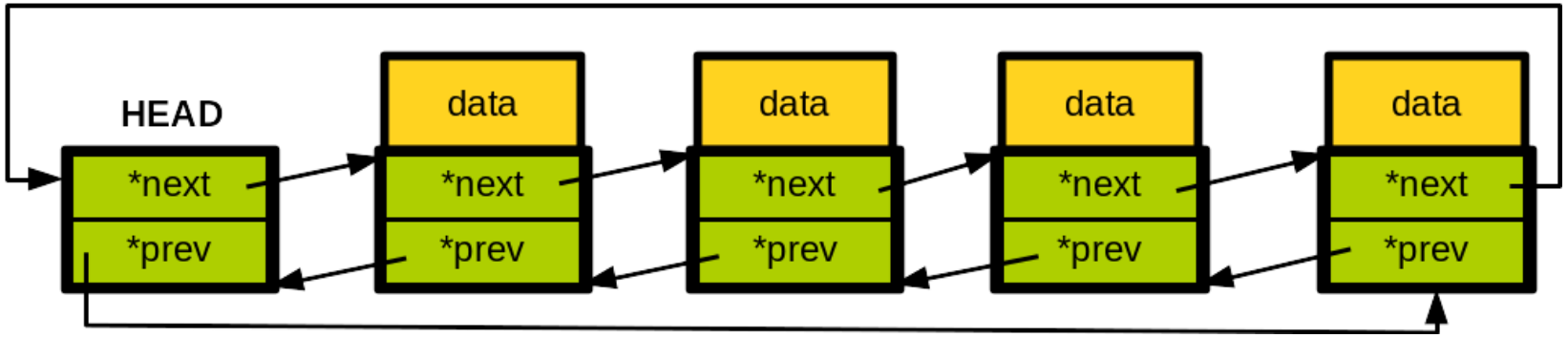
Common design patterns for kernel data structures

- **The Toolbox philosophy**: Building blocks over complete solutions
  - *No one-size-fits-all solutions*
  - *Only low-level primitives and tools to build custom, efficient solutions*
    - *Specialized for developers' specific needs*
- Kernel provides basic building blocks, not complete implementations
  - E.g., very basic search macro
- Flexibility through composition
  - Mix and match data structures
  - No forced memory allocation or ownership model

Common design patterns of kernel data structures

- Zero cost abstraction
  - Macros and inline functions to avoid function call overhead
  - Compile-time type checking to avoid runtime penalties
  - `container_of()` for type-safe casting
  - No hidden allocations
- Synchronization is done by the user, not by the data structure

The canonical example: Linked list (doubly)



- Starts from **HEAD** and terminates at **HEAD**
- When empty, **HEAD** is not **NULL**
  - **prev** and **next** of **HEAD** points to **HEAD**
  - **HEAD** is a sentinel node
- Easy to insert a new element at the end of the list
- There is no exceptional case to handle **NULL**

The canonical example: Linked list (doubly)

- A circular doubly linked list
- Two differences from a typical design
  - Embed a linked list node in the structure
  - Use the sentinel node as a list header
    - Removes NULL checks and edge cases

- **`include/linux/list.h`**

Linux linked list declaration example

```
struct list_head {                /* kernel linked list data structure */
  struct list_head *next, *prev;
};

struct car {
  unsigned int max_speed;
  unsigned int drive_wheel_num;
  unsigned int price_in_francs;
  struct list_head list;          /* add list_head instead of prev and next */
};

struct list_head my_car_list;     /* HEAD is also list_head */
```

- **`struct list_head`** is the list data structure
- **`list_head`** is embedded in the data structure (**`struct car`**)
- The list starts from **`list_head`**; **`my_car_list`** → sentinel node

Defining a list

```
struct car *my_car = kmalloc(sizeof(*my_car), GFP_KERNEL);
my_car->max_speed = 150;
my_car->drive_wheel_num = 2;
my_car->price_in_francs = 10000;
INIT_LIST_HEAD(&my_car->list); /* initialize an element */

struct car my_car {
    .max_speed = 150,
    .drive_wheel_num = 2,
    .price_in_francs = 10000,
    .list = LIST_HEAD_INIT(my_car.list), /* initialize an element */
}

LIST_HEAD(my_car_list); /* initialize the HEAD of a list */
```

- Initialize **`list_head`**
- **`list_head→prev = &list_head;`**
- **`list_head→next = &list_head;`**

Accessing your data: `container_of`

Q. How to get a pointer to a **`struct car`** from its **`list`**?

- use **`list_entry(ptr, type, member)`**
- Just a pointer arithmetic

```
struct car *amazing_car = list_entry(&my_car_list, struct car, list);

/**
* list_entry - get the struct for this entry
* @ptr: the &struct list_head pointer.
* @type: the type of the struct this is embedded in.
* @member: the name of the list_head within the struct.
*/
#define list_entry(ptr, type, member) container_of(ptr, type, member)
#define container_of(ptr, type, member) ({                              \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);                \
    (type *)( (char *)__mptr - offsetof(type,member) );})
#define offsetof(TYPE, MEMBER) ((size_t)&((TYPE *)0)->MEMBER)
```

Let's break it down!

```
Offset  Size  Field
------  ----  ----------------------------------------
0x1000    4     max_speed
0x1004    4     drive_when_need
0x1008    4     price_in_francs
0x100C    4     (padding for 8-byte alignment)
0x1010    8     list.next     (pointer)
0x1018    8     list.prev     (pointer)
------  ----
```

- `offsetof(struct car, list) == 0x10 (16)`
- `sizeof(struct car) == 32`

Iterating over a list: O(n)

```
/**
* list_for_each - iterate over a list
* @pos: the &struct list_head to use as a loop cursor.
* @head: the head for your list.
*/
#define list_for_each(pos, head)                                    \
    for (pos = (head)->next; pos != (head); pos = pos->next)

/**
* list_for_each_entry - iterate over list of given type
* @pos: the type * to use as a loop cursor.
* @head: the head for your list.
* @member: the name of the list_head within the struct.
*/
#define list_for_each_entry(pos, head, member)                      \
    for (pos = list_first_entry(head, typeof(*pos), member);        \
        &pos->member != (head);                                     \
        pos = list_next_entry(pos, member))
```

returns `list_head` nodes

returns `struct car *` nodes

Iterating over a list: O(n)

```c
/* Temporary variable needed to iterate: */
struct list_head p;

/* This will point on the actual data structures
* (struct car)during the iteration: */
struct car *current_car;

list_for_each(p, &my_car_list) {
    current_car = list_entry(p, struct car, list);
    printk(KERN_INFO "Price: %lf\n", current_car->price_in_francs);
}

/* Simpler: use list_for_each_entry */
list_for_each_entry(current_car, &my_car_list, list) {
    printk(KERN_INFO "Price: %lf\n", current_car->price_in_francs);
}
```

returns `list_head` nodes

returns `struct car *` nodes

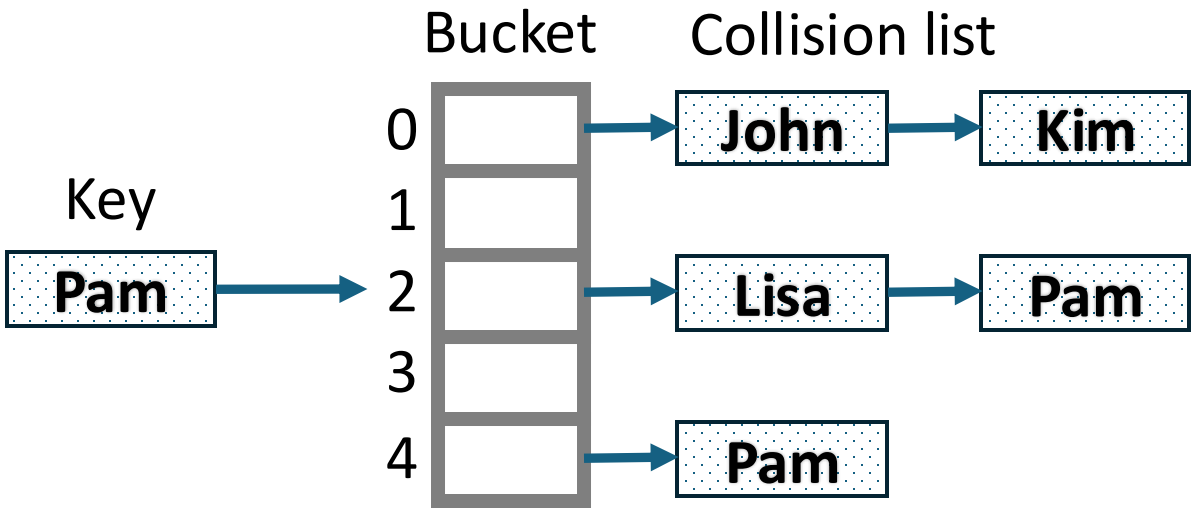
- Backward iteration?
  - **`list_for_each_entry_reverse(pos, head, member)`**

Linked list usage in the kernel

- Kernel code makes extensive use of linked lists:
  - A list of threads under the same parent PID
  - A list of superblocks of a file system
  - and many more …

Linux hash table

- A simple fixed-size open chaining hash table
  - The size of the bucket is fixed at the initialization time as 2^N
  - Each bucket has a singly linked list to resolve collision
  - Time complexity: O(1)

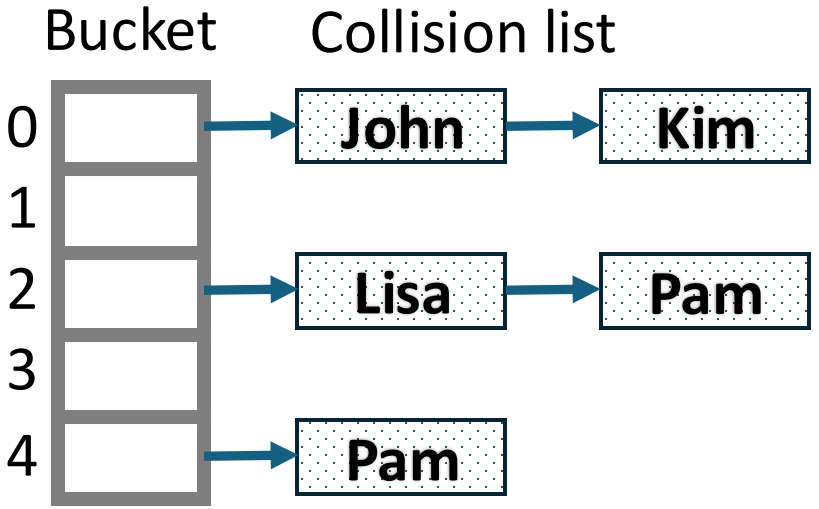


Linux hash table structure and declaration

```c
/* linux/include/linux/hashtable.h, types.h */
/* hash bucket */
struct hlist_head {
    struct hlist_node *first;
};

/* collision list */
struct hlist_node {
    /* Similar to list_head, hlist_node is embedded
     * into a data structure.
     */
    struct hlist_node *next;
    struct hlist_node **pprev; /* &prev->next */
};

/**
* Define a hashtable with 2^bits buckets
*/
#define DEFINE_HASHTABLE(name, bits) ...
#define hash_init(hashtable) ...
#define hash_add(hashtable, node, key) ...
#define hash_for_each_possible(name, obj, member, key) ...
void hash_del(struct hlist_node *node);
```

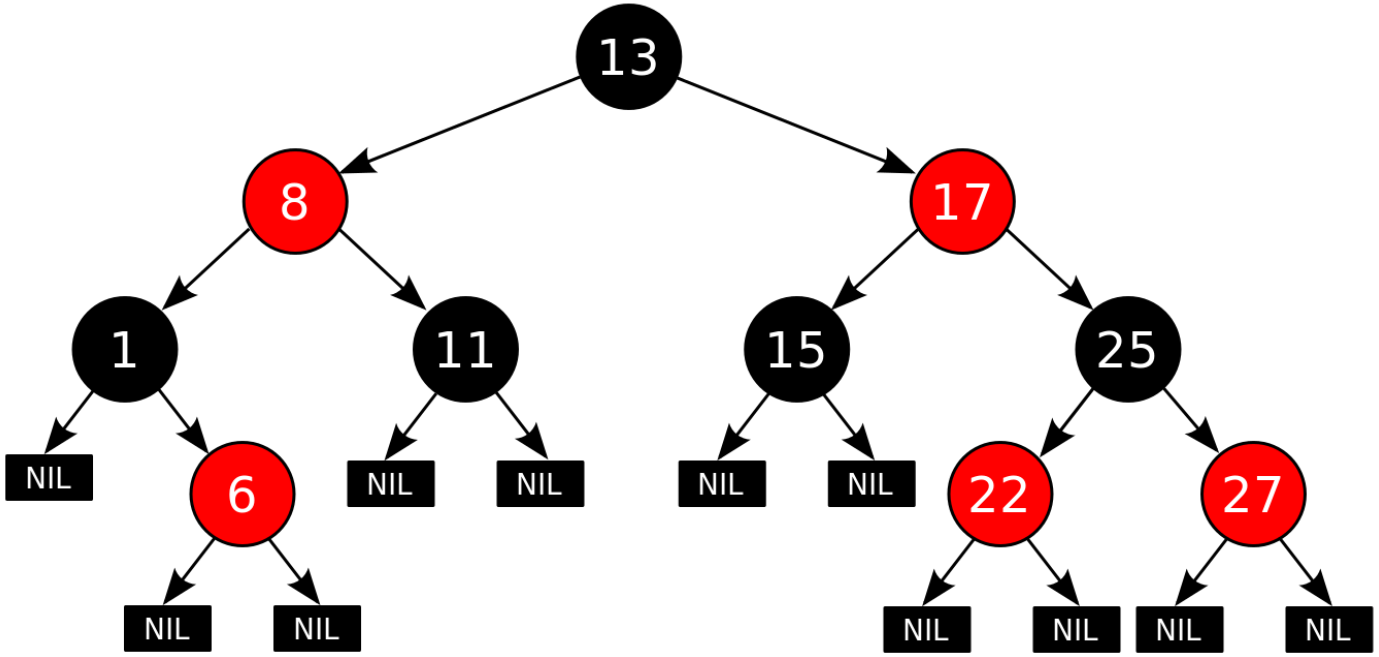


Linux hash table use in the kernel

- Transparent hugepages
  - Finds physically consecutive 4KB pages
  - Remaps consecutive 4KB pages to a 2MB page (hugepage)
  - Saves TLB entries and improves memory access performance by reducing TLB misses
  - Maintains per-process memory structure (**`struct mm_struct`**)
- Also used by kernel same page merging (KSM)

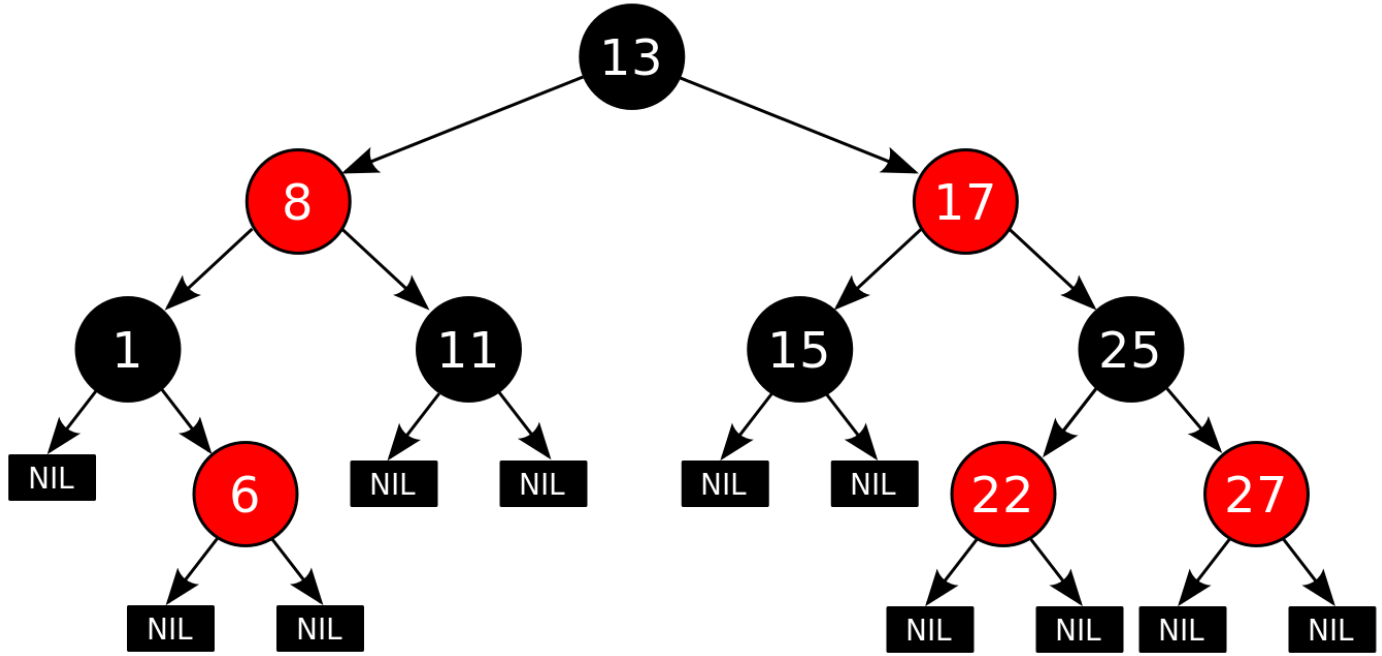
Tree basics: Red-black tree

- Provides ordered sets with fast search, insert, and delete
- A type of self-balancing binary search tree
  - Nodes: red or black
  - Leaves: black, no data



Tree basics: Red-black tree

- Following properties are maintained during tree ops:
  - Path from a node to its NIL node has the same number of black nodes (black-height)
- Fast search, insert, delete operations: $O(\log^N)$



Linux red-black tree example

- EEVDF scheduler
  - Default task scheduler in Linux
  - Combines proportional-share scheduling with latency bounds
- Principles:
  - Schedule task with the **earliest virtual deadline** among all **eligible** tasks
  - Two-dimensional scheduling decision: eligibility + deadline
  - Guarantees that no task waits longer than O(n*slice) time
- Each task has **a `vruntime`** → how much the task has virtually run
- Per-task **`vruntime`** is maintained in the rbtree

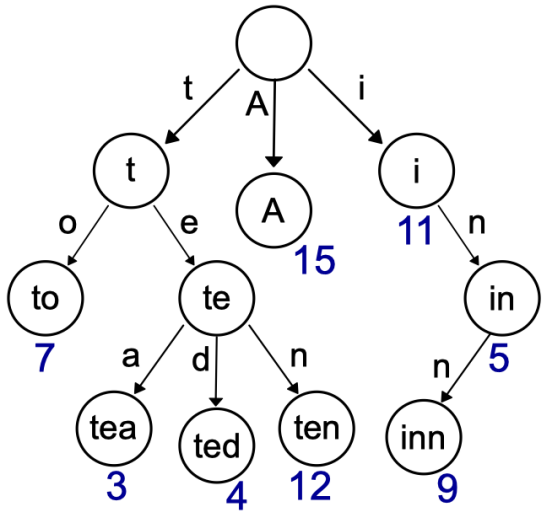
Focus of today's lecture

- Kernel data structures
  - Linked list
  - Hash table
  - Red-black tree
  - **Xarray**
  - **Maple tree**
- Handling concurrency
  - Atomics
  - Spinlock and its readers-writer variant
  - Mutex and its readers-writer variant

Deviates from its own toolbox philosophy

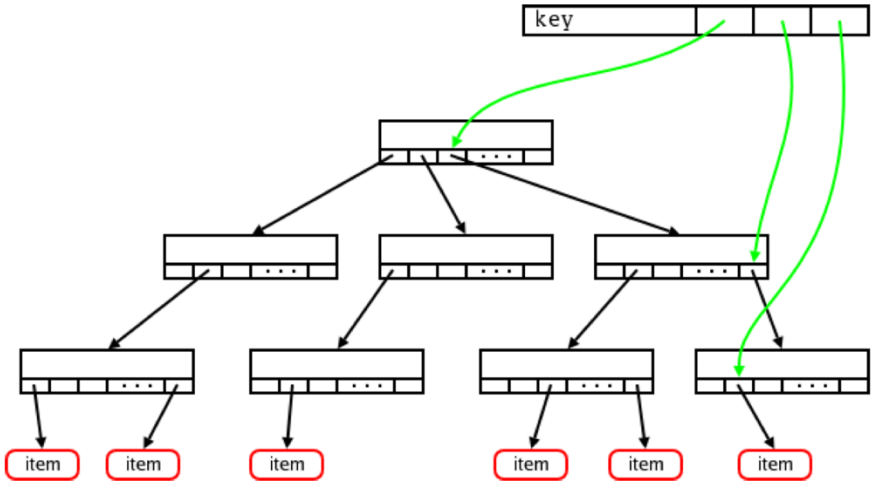
Radix tree (or trie)

- Prefix-based organization
  - Autocomplete, spell checking, IP routing
  - No hash collision unlike hash tables
- The key at each node is compared **chunk-of-bits by chunk-of-bits**
- All descendants of a node have a common prefix
- Values are only associated with leaves
- Source: Wikipedia



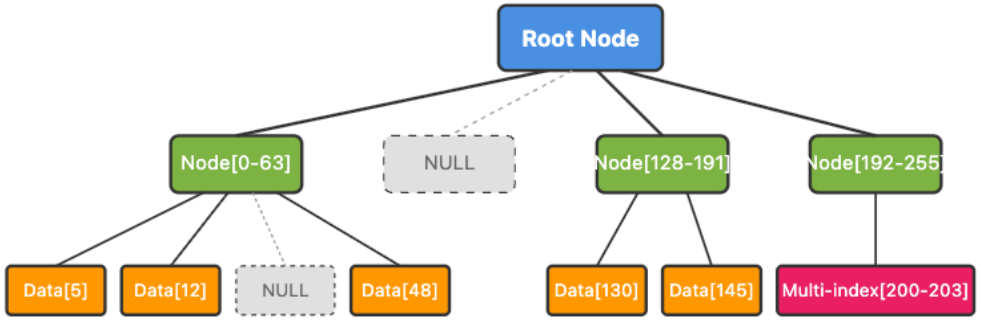
Linux radix tree (basis of XArray)

- Maps **`unsigned long`** to **`void *`**
- Each node has 64 slots
- Slots are indexed by a 6-bit ($2^6 = 64$) portion of the key
- **Leaf** → a slot points to an address of data
- **Non-leaf** → a slot points to another node in a lower layer
- Other metadata is stored per node:
  - **tags**
- Source: LWN



eXtensible Array (XArray)

- Multi-level filing system for a massive, mostly empty warehouse (key → value)
  - Nice wrapper around radix tree
  - Sparse representation (memory efficient)
  - Lockless lookups with RCU
  - Support tags for quick searches
  - Multi-index support



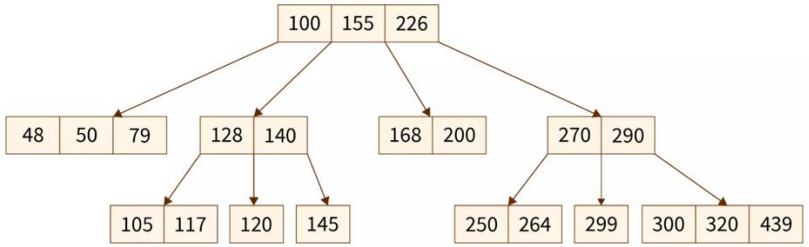
Let's try to understand the basic radix tree to get an idea of how it works!

XArray use in the kernel

- **Page cache**
  - For every page lookup in a file, check xarray to see if it is cached
  - Use tags to maintain the status of the page (e.g., PAGECACHE_TAG_DIRTY or PAGECACHE_TAG_WRITEBACK)
- **File descriptors**
  - XArray maps fd number to the internal struct file (information about open file)
- **I/O polling**
  - io_uring uses XArrays to map its tables of files and other resources

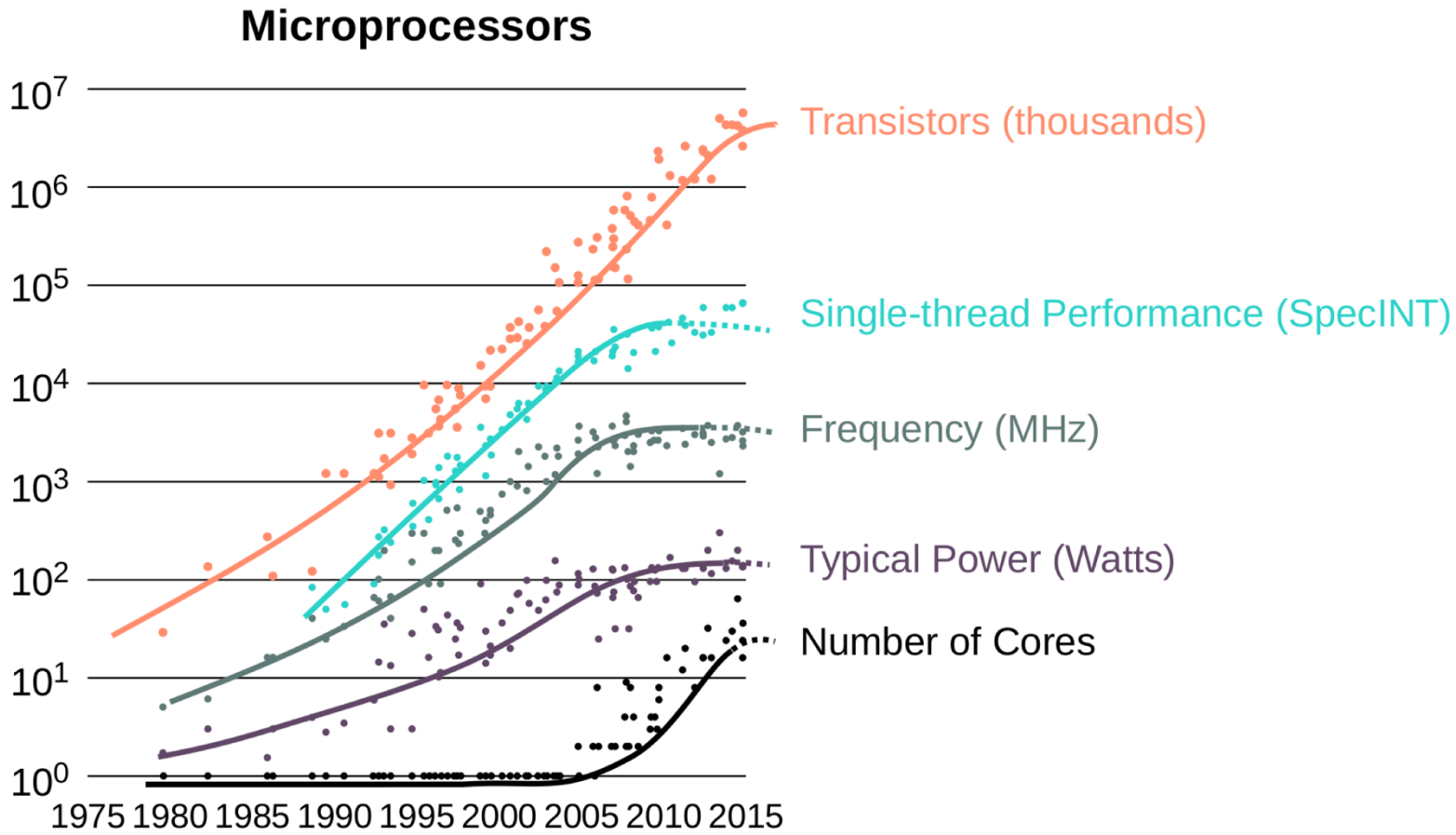
Maple tree

- A B-tree variant data structure
  - Efficient **range lookups**
  - Fast iteration
  - Better scalability compared to rbtree
- Optimized for dense and sparse data
- Lockless operations using RCU
  - Addresses read-heavy cases
- Users of the kernel
  - VMA in the kernel (replaced rbtree)



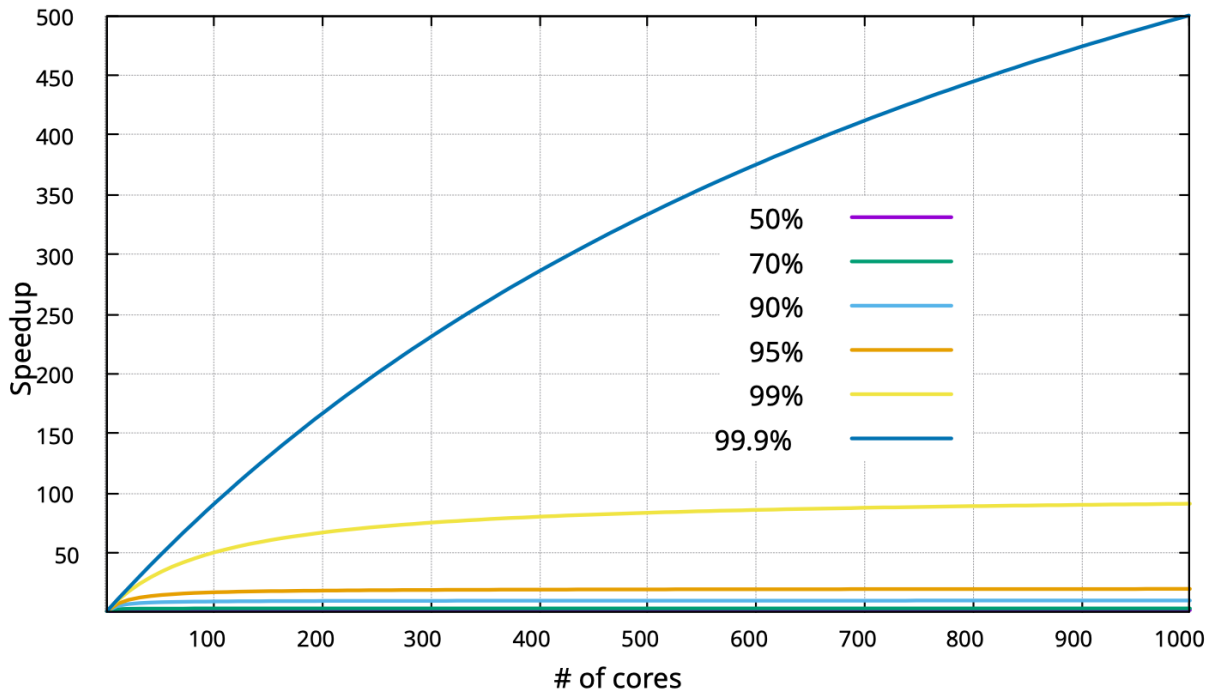
Concurrency in the Kernel: Protecting Shared Data

The end of the free lunch: single core scaling stopped!



The tyranny of the sequential part

- **Amdahl's law: theoretical speedup of the task execution**
  - Speedup = **`1/(1 - p + p/n)`**
  - **`p:`** parallel portion of a task
  - **`n:`** number of CPUs



Where are such sequential parts

- Applications: sequential algorithms
- Libraries: memory allocator (buddy structure)
- OS kernel
  - Memory management: VMA (virtual memory area)
  - File system: file descriptor table, journaling
  - Network stack: receive queue

Kernel concurrency and synchronization

- The kernel is programmed using the shared memory model
- **Critical section (also called critical region)**
  - Code paths that access and manipulate shared data
  - Must execute **atomically** without interruption
  - Should not be executed in parallel → must be always sequential!

- **Race condition**
  - Two threads concurrently executing the same critical region → Bug!

Causes of concurrency in the kernel

- **Sleeping and synchronization with user space**
  - A task in the kernel can sleep → invokes the scheduler → new process gets scheduled
- **Interrupts [higher priority]**
  - An interrupt can occur asynchronously at almost any time → interrupts the currently executing code
- **Softirqs and tasklets [higher priority]**
  - Kernel can schedule its own higher-priority work at almost any time, interrupting the currently executing code

Concurrency safety

- **SMP** (symmetric multiprocessing) **safe**
  - Code that is safe from concurrency on multicore machines
- **Preemption safe**
  - Code that is safe from concurrency with kernel preemption
- **Interrupt safe**
  - Code that is safe from concurrent access from an interrupt handler

The race condition: updating a single variable

```
int i;
void foo(void)
{
  i++;
}
```

Q. What happens if two threads concurrently execute `foo()`?

Q. What happens if two threads concurrently update `i`?

Q. Is incrementing `i` an atomic operation?

The race condition: updating a single variable

A single statement

```
/* C code */
  i++;
```

It can translate into multiple machine instructions

```
/* Machine instructions */
01: get the current value of i and copy it into a register
02: add one to the value stored in the register
03: write back to memory the new value of i
```

Let’s check what happens if two threads concurrently update **i**

The race condition: updating a single variable

- Two threads are running, the initial value of **`i`** is **7**

Thread 1	Thread 2
`get i(7)`	-
`increment i(7 → 8)`	-
`write back i(8)`	-
-	`get i(8)`
-	`increment i (8 → 9)`
-	`write back i (9)`

- As expected, **7** incremented twice is **9**

The race condition: updating a single variable

- Two threads are running, the initial value of **`i`** is **7**

Thread 1	Thread 2
`get i(7)`	`get i(7)`
`increment i(7 → 8)`	-
-	`increment i (7 → 8)`
`write back i(8)`	-
-	`write back i (8)`

- If both threads read the value of initial value of **`i`** before it is incremented, both threads should increment and save the same value

Solution: atomic instruction

- Two threads are running, the initial value of **`i`** is **7**

Thread 1	Thread 2
`increment & store i (7 → 8)`	-
-	`increment & store i (8 → 9)`

Thread 1	Thread 2
-	`increment & store i (7 → 8)`
`increment & store i (8 → 9)`	-

- Hardware provides the atomicity guarantee
  - Impossible for two atomic operations to interleave (for the same object)

Atomic operations

- Examples:
  - **`fetch-and-add`**: Atomic increment
  - **`test-and-set`**: set a value at a memory location and return the previous value
  - **`compare-and-swap`**: modify the content of a memory location only if the previous content is equal to a given value
- Linux provides two APIs:
  - Integer atomic operations
  - Bitwise atomic operations

Q. Are atomic operations enough to perform complex operations in any program?

Locking

- Atomic operations are insufficient for protecting shared data in long and complex critical regions
  - E.g., updating a virtual memory area, updating a page cache for an inode

- **Require only one thread to manipulate the data structure at a time**
  - A mechanism for preventing access to a resource while another thread is already in that marked region → **lock**

Locking issues: deadlocks

- Situations in which one or several threads are waiting on locks for one or several resources that will never be freed
  - None of the threads can continue
- **Self-deadlock**
  - **Note:** Linux does not support **recursive locking**

```
acquire lock
acquire lock, again
wait for lock to become available
...
```

Deadlock in more detail

- Deadly embrace (ABBA problem)

Thread 1	Thread 2
`acquire lock A`	`acquire lock B`
`try to acquire lock B`	`try to acquire lock A`
`wait for lock B`	`wait for lock A`

Deadlock prevention: lock ordering

- Always acquire **locks in an ordered manner**

1

```
/* linux/mm/filemap.c */
/*
 * Lock ordering:
 *
 *  ->i_mmap_rwsem      (truncate_pagecache)
 *    ->private_lock        (__free_pte->__set_page_dirty_buffers
 *      ->swap_lock     (exclusive_swap_page, others)
 *        ->mapping->tree_lock
 *
 *  ->i_mutex
 *    ->i_mmap_rwsem        (truncate->unmap_mapping_range)
 *  ...
 */
```

2

```
/* linux/fs/namei.c */
struct dentry *lock_rename(struct dentry *p1, struct dentry *p2)
{
    struct dentry *p;
    if (p1 == p2) {
        inode_lock_nested(p1->d_inode, I_MUTEX_PARENT);
        return NULL;
    }
    mutex_lock(&p1->d_sb->s_vfs_rename_mutex);
    p = d_ancestor(p2, p1);
    if (p) {
        inode_lock_nested(p2->d_inode, I_MUTEX_PARENT);
        inode_lock_nested(p1->d_inode, I_MUTEX_CHILD);
        return p;
    }
    p = d_ancestor(p1, p2);
    if (p) {
        inode_lock_nested(p1->d_inode, I_MUTEX_PARENT);
        inode_lock_nested(p2->d_inode, I_MUTEX_CHILD);
        return p;
    }
    inode_lock_nested(p1->d_inode, I_MUTEX_PARENT);
    inode_lock_nested(p2->d_inode, I_MUTEX_PARENT2);
    return NULL;
}
```

Lock design & usage: contention and scalability

- **Lock contention:** a lock is currently in use, but another thread is trying to acquire
- **Scalability:** how well a system can be expanded with a large number of processors
- **Coarse- vs. fine-grained locking**
  - Coarse-grained locks: bottleneck on high-core count machines
  - Fine-grained locks: overhead on low core count machines

Lock type: Spinlocks (the busy-wait lock)

- Most common lock used in the kernel
- When a thread tries to acquire an already held lock, it **spins** while waiting for the lock to become available
  - Wasting processor time when spinning is too long
  - Spinlocks can be used in an interrupt context, in which a thread cannot sleep
    → Kernel provides a special spinlock API for data structures shared in the **interrupt context**
- In the process context, do not sleep while holding a spinlock
  - Preemption is disabled → tasks waiting/locked cannot move to another core

Spinlock issue: working with interrupts

```
/* include/linux/spinlock.h */
DEFINE_SPINLOCK(my_lock);

spin_lock(&my_lock);
/* critical region */
spin_unlock(&my_lock);
```

- **`spin_lock()`** is not recursive → self-deadlock
- **`lock/unlock`** methods disable/enable kernel preemption and acquire/release the lock
  - Kernel cannot schedule other tasks on the CPU where lock/unlock code runs
- Lock is compiled away on uniprocessor systems
  - Still need preemption APIs to prevent task interleaving

Spinlock in interrupt handler

- Spin locks do not sleep → safe to use in an interrupt context

- If a lock is used in an interrupt handler, **must disable local interrupts** before obtaining the lock
  - Otherwise, an interrupt handler can interrupt kernel code while the lock is held and attempt to reacquire the lock
  - The interrupt handler spins, waiting for the lock to become available
    - The lock holder, however, does not run until the interrupt handler completes → **double-acquire deadlock**

What is the issue with this code?

```
DEFINE_HASHTABLE(global_hashtbl, 10);
DEFINE_SPINLOCK(hashtbl_lock);

irqreturn_t irq_handler(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock(&hashtbl_lock);
    /* access global hashtbl */
    spin_unlock(&hashtbl_lock);
}

int foo(void)
{
    /* A function running in process context */
    spin_lock(&hashtbl_lock);
    /* What happens if an interrupt occurs
    * while a task executing here? -> Deadlock */
    spin_unlock(&hashtbl_lock);
}
```

Bugfix for the deadlock case: Enable/disable interrupts

```c
DEFINE_HASHTABLE(global_hashtbl, 10);
DEFINE_SPINLOCK(hashtbl_lock);

irqreturn_t irq_handler(int irq, void *dev_id)
{
    /* Interrupt handler running in interrupt context */
    spin_lock(&hashtbl_lock);
    /* It is okay NOT to disable interrupt here
     * because this is the only interrupt handler
     * access the shared data and this particular
     * interrupt is already disabled.
     */
    spin_unlock(&hashtbl_lock);
}

int foo(void)
{
    /* A function running in process context */
    spin_lock_irqsave(&hashtbl_lock);
    /* What happens if an interrupt occurs
    * while a task executing here? -> Deadlock */
    spin_unlock_irqrestore(&hashtbl_lock);
}
```

Lock design: Mutexes (the sleeping lock)

- Mutex is a sleeping lock
  - Only one thread can hold the mutex at a time
  - A thread locking a mutex must unlock it
  - No recursive lock and unlock operations
  - A thread cannot exit while holding a mutex
  - A mutex cannot be acquired in an interrupt context
  - A thread holding a mutex can also sleep

- Other threads that wait for the mutex
  - Pushed to the wait queue and put to sleep
  - One sleeping thread is woken up once the current holder releases the mutex

Spinlock vs. mutex: The tradeoff

Requirement	Recommended lock
Low overhead locking	spinlock is preferred
Short lock hold time	spinlock is preferred
Long lock hold time	mutex is preferred
Need to lock from the interrupt context	spinlock is required
Fine with sleeping when holding the lock	mutex is required

Synchronization golden rule

Key Takeaways:

- **Rule #1:** Protect **data**, not code.
- **Rule #2:** Never sleep while holding a **spinlock**.
- **Rule #3:** If interrupts are involved, use **spin_lock_irqsave()**.
- **Rule #4:** Choose the simplest synchronization primitive that works first (**correctness first, performance second**).

Questionnaire for locking

Question	Implication	Locking needed?
Is the data global or shared?	Can be accessed from multiple contexts	Yes
Can another thread or CPU access it?	Possible concurrent access	Yes
Is data accessed from both process context and interrupt context?	Locks supporting interrupt context	spinlock
Is data shared between interrupt handlers?	Must handle concurrent interrupt execution	spinlock + interrupts disable/enable
If a process is preempted while using the data, can another process access it?	Ensure data consistency across context switches	Mutex
If the current process sleeps while holding data, what state is the data left in?	If data is shared, it will be inconsistent; need to use locks	Mutex
What if a function is called concurrently on another CPU?	Need to ensure data consistency if 2 CPUs work on the same data	Yes

Further readings

- LWN: Trees I: Radix trees
- Bit arrays and bit operations in the Linux kernel
- LWN: The XArray data structure
- The design and implementation of the Xarray
- Red-black Trees (rbtree) in Linux