CSE 511: Operating Systems Design

Lecture 27

Assignment 3 Questions

Epoch-Based Reclamation

File Systems

Last Lecture (before Thanksgiving)

- We talked about *file systems*
  - File meta data
  - Mapping files to blocks
    - Linked lists
    - Indexed table
    - Direct, indirect, double-indirect, triple-indirect mappings
  - Tracking of free/allocated space
    - Free lists (slow), bitmap, list of extents (trees)

Today

- *First half:* Assignment 3
  - Question posted on Slack regarding M&S queue
  - Epoch-Based Reclamation
    - We provided an implementation (smr.h)
    - Clarify some misconceptions that I have noticed thus far
- *Second half:* Will continue discussing our previous topic (file systems)

Please submit SRTE evaluation

- Please submit the SRTE evaluation for CSE 511
  - You should have received the notification from the SRTE system
  - Your feedback is very important!
  - Should be available **till Dec 11**

Assignment 3 Questions

- Get the latest version of the assignment and lecture slides
  - Clarification, extra links, and some typos fixed
- Define _Atomic *only* for **shared** variables
    - Old and New values in CAS are still regular variables (they are local, not shared)
    - Use brackets **_Atomic(struct node *)** top rather than _Atomic struct node * top
    - Pointer to an atomic variable vs. an atomic variable containing a pointer (and both)

Will M&S queue ever try to dereference next = NULL?

- An interesting question posted on Slack
  - Questions like this are useful for better understanding of the algorithms
- *Example:* Consider an *empty* queue (head = tail = dummy node). dequeue() retrieves the current head (h), and then a concurrent enqueue() inserts another element such that *h->next* is modified (i.e., no longer **NULL**). <u>Is it possible that dequeue() retrieves a stale value (**NULL**) for **next** (because the queue was initially empty) when it attempts to return the element that was just inserted?</u>

Michael-Scott's Lock-Free Queue

```
void *dequeue() {
    while (true) {
        node_t *h = LOAD(head);
        node_t *t = LOAD(tail);
        node_t *next = LOAD(h->next);
        if (h == LOAD(head)) {
            if (h == t) {
                if (next == NULL) return NULL; // Empty
                CAS(&tail, t, next);
            } else {
                void *obj = next->value; // Will it SEGFAULT here?
                if (CAS(&head, h, next)) {
                    // Free old sentinel (`h'): can still be used by other
                    // threads; `next' is now the new sentinel node
                    return obj; // Return the object
                }
}   }   }   }
```

Michael-Scott's Lock-Free Queue

```
void *dequeue() {
    while (true) {
        node_t *h = LOAD(head);
        node_t *t = LOAD(tail);
        node_t *next = LOAD(h->next);
        if (h == LOAD(head)) { // That will not change by enqueue()
            if (h == t) { // The queue is empty (is it still empty?)
                if (next == NULL) return NULL; // Empty
                CAS(&tail, t, next);
            } else {
                void *obj = next->value; // Will it SEGFAULT here?
                if (CAS(&head, h, next)) {
                    // Free old sentinel (`h'): can still be used by other
                    // threads; `next' is now the new sentinel node
                    return obj; // Return the object
                }
}   }   }   }
```

Michael-Scott's Lock-Free Queue

```
void enqueue(void *obj) {
    node_t *node = malloc(sizeof(node_t);
    node->obj = obj;
    node->next = NULL;
    while (true) {
        node_t *t = LOAD(tail);
        node_t *next = LOAD(t->next);
        if (t == LOAD(tail)) {
            if (next == NULL) {
                if (CAS(&t->next, next, node)) break; // 1st step
            } else {
                CAS(&tail, t, next); // Help to move tail - 2nd step
            }
        }
    }
    CAS(&tail, t, node); // Try to move tail (one time) - 2nd step
}
```

Will M&S queue ever try to dereference next = NULL?

- Observation 1: enqueue() modifies **next** before modifying **tail**
- Observation 2: dequeue() loads **tail** before loading **next**
- Two possibilities exist for dequeue()
  - **next** is still **NULL** but then local(**head**) = local(**tail**) because we read local(**tail**) before **next**
  - local(**head**) != local(**tail**): means enqueue() has already moved the tail but then **next** != **NULL** since **next** is modified before tail in enqueue()

Michael-Scott's Lock-Free Queue

```
void *dequeue() {
    while (true) {
        node_t *h = LOAD(head);
        node_t *t = LOAD(tail);
        node_t *next = LOAD(h->next);
        if (h == LOAD(head)) {
            if (h == t) { // 1st case: next can be NULL
                if (next == NULL) return NULL; // Empty
                CAS(&tail, t, next);
            } else {
                void *obj = next->value; // 2nd case: next != NULL
                if (CAS(&head, h, next)) {
                    // Free old sentinel (`h'): can still be used by other
                    // threads; `next' is now the new sentinel node
                    return obj; // Return the object
                }
}   }   }   }
```

The order of these loads is very important!

Example of an erroneous implementation

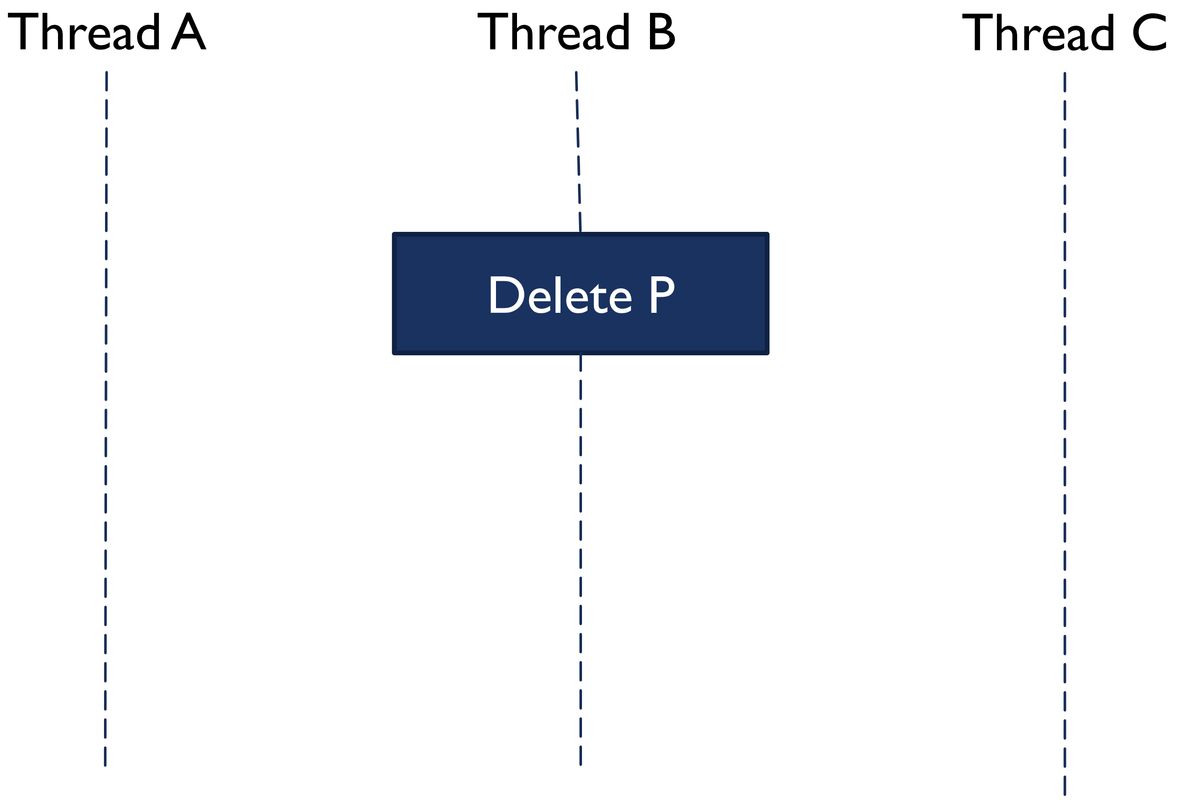
BUG (race condition)!!!
next can become NULL while h != t

```
void *dequeue() {
    while (true) {
        node_t *h = LOAD(head);
        node_t *next = LOAD(h->next);
        node_t *t = LOAD(tail);
I       if (h == LOAD(head)) {
            if (h == t) {
                if (next == NULL) return NULL; // Empty
                CAS(&tail, t, next);
            } else {
                void *obj = next->value; // Can SEGFAULT here
                if (CAS(&head, h, next)) {
                    // Free old sentinel (`h’): can still be used by other
                    // threads; `next’ is now the new sentinel node
                    return obj; // Return the object
                }
}   }   }   }
```

Epoch-Based Reclamation (EBR)

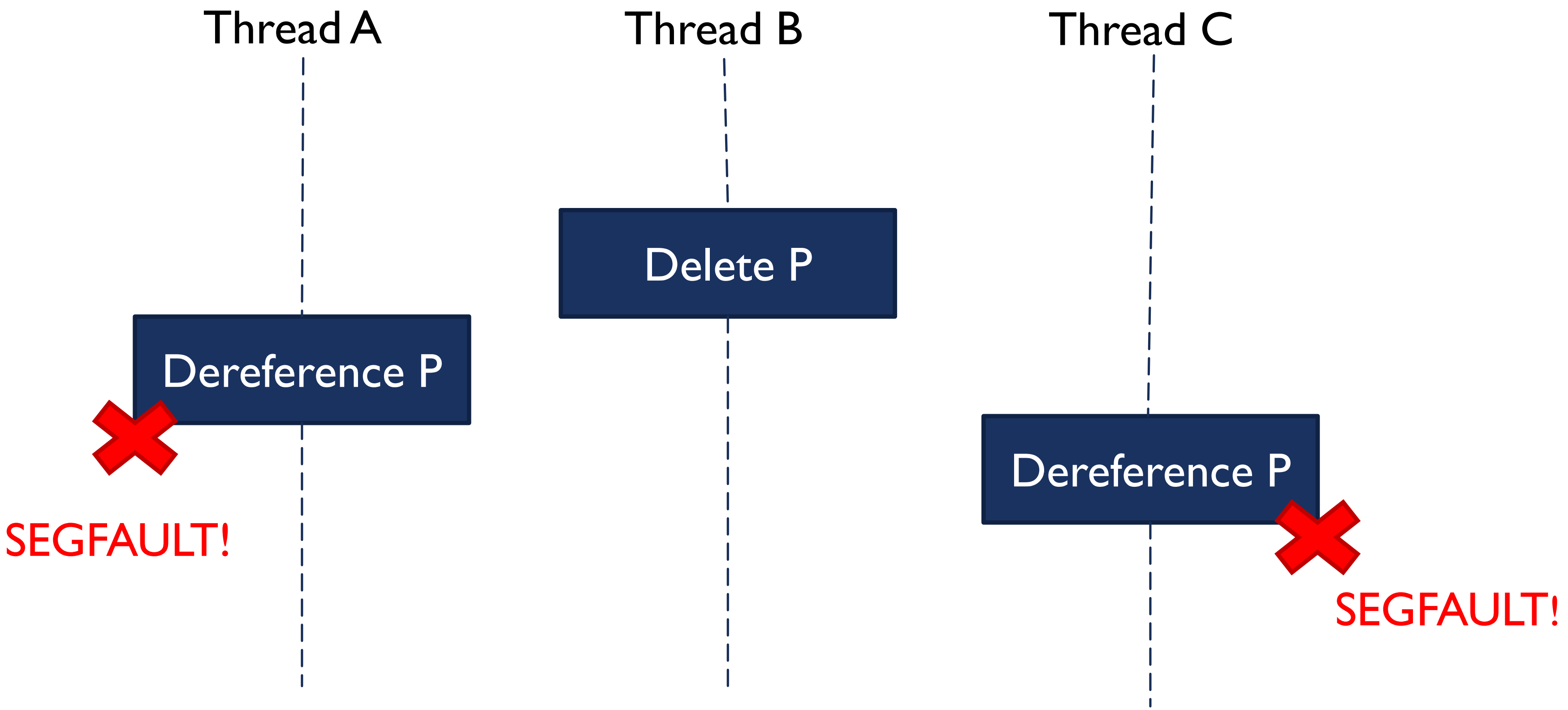
- You do not need to implement it
  - We provided the code in smr.h and smr.c
- I have seen some examples of incorrect usage, e.g., *separate modifications* are wrapped by begin_op() and end_op() rather than the *entire operation*
  - The entire **push** (**enqueue**) or **pop** (**dequeue**) operation has to be wrapped by these
  - It is possible to optimize certain corner cases (e.g., in Treiber's stack) but **avoid** it for this assignment because it can be error-prone

MEMORY RECLAMATION PROBLEM



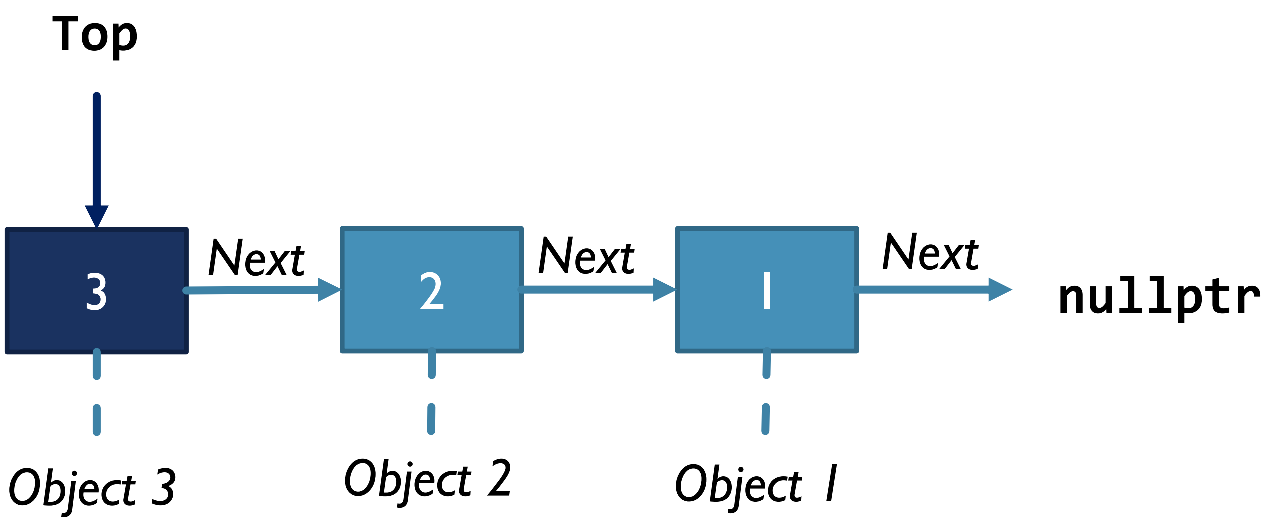
One thread wants to de-allocate a memory block which is still reachable by concurrent threads

MEMORY RECLAMATION PROBLEM



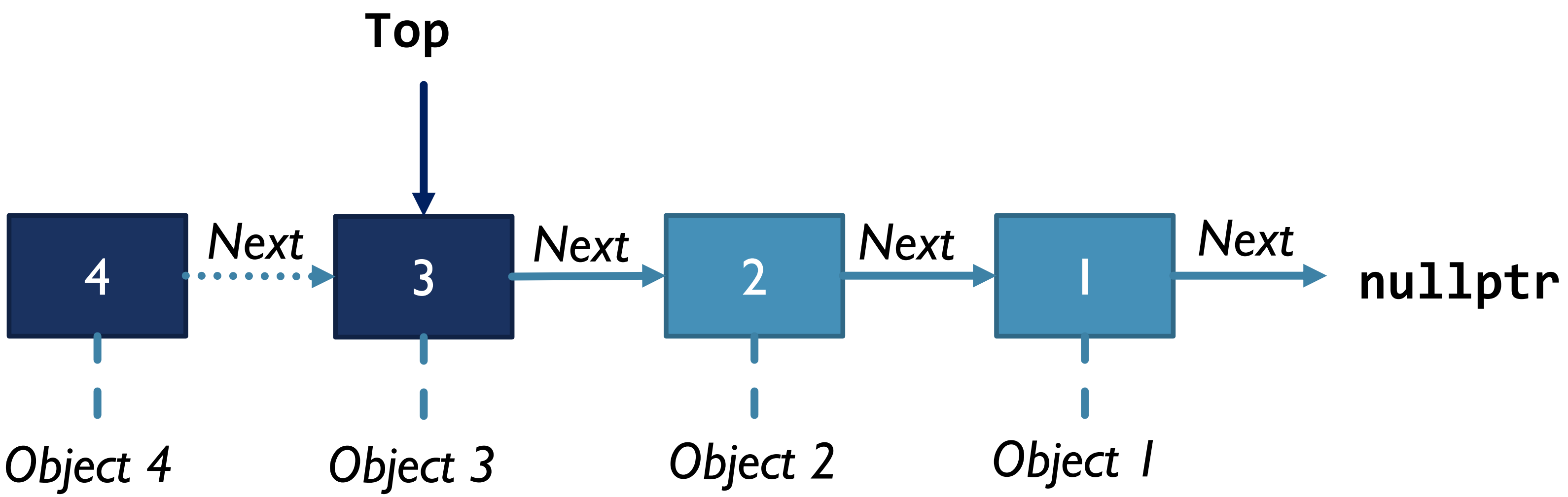
One thread wants to de-allocate a memory block which is still reachable by concurrent threads

TREIBER'S LOCK-FREE STACK



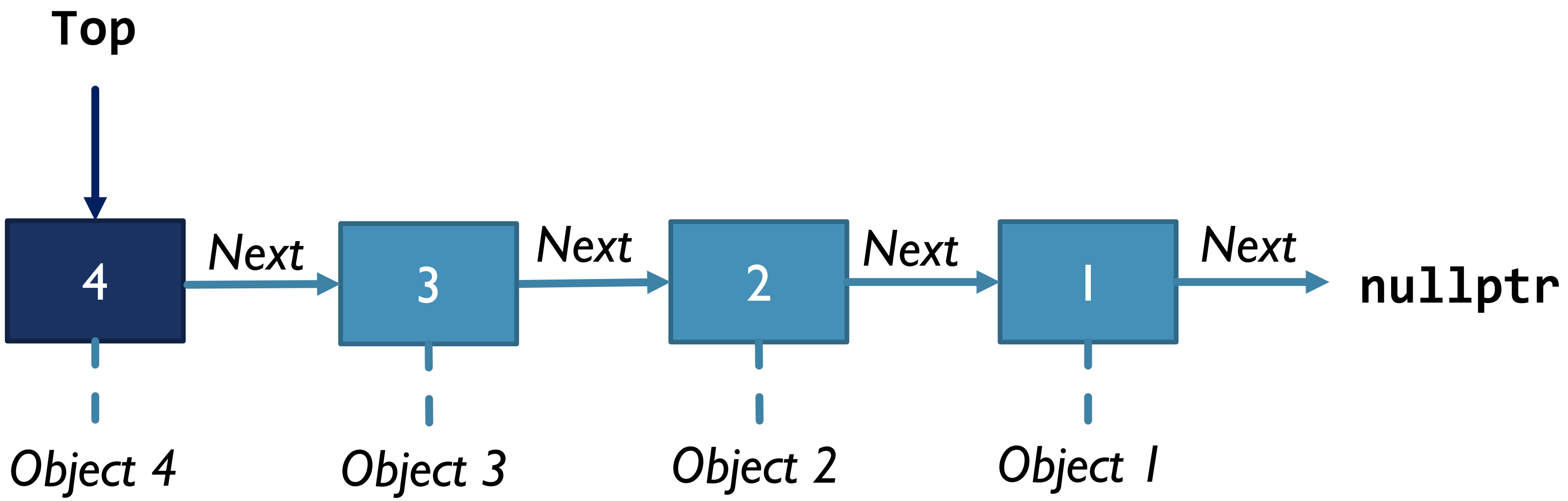
- PUSH and POP operations are implemented by updating **Top** using CAS

TREIBER'S LOCK-FREE STACK



- PUSH and POP operations are implemented by updating **Top** using CAS

TREIBER'S LOCK-FREE STACK



- PUSH and POP operations are implemented by updating **Top** using CAS

EXAMPLE: NO RECLAMATION

```
struct Node {
   Node* next;    // Next element
   Object* obj;  // Stored object
};
Node* Top = nullptr;
```

EXAMPLE: NO RECLAMATION

```
struct Node {
   Node* next;    // Next element
   Object* obj;   // Stored object
};
Node* Top = nullptr;

PUSH(Object* obj) {
   Node* node = malloc(…);
   node->obj = obj;
   while (true) {
      node->next = Top;
      if (CAS(&Top, node->next, node))
         break;
   }
}
```

EXAMPLE: NO RECLAMATION

```
struct Node {
    Node* next;    // Next element
    Object* obj;   // Stored object
};
Node* Top = nullptr;

PUSH(Object* obj) {
    Node* node = malloc(…);
    node->obj = obj;
    while (true) {
        node->next = Top;
        if (CAS(&Top, node->next, node))
            break;
    }
}
```

```
Object* POP() {
    Object* obj = nullptr;
    while (true) {
        Node* node = Top;
        if (node == nullptr)
            break;
        if (CAS(&Top, node, node->next) {
            obj = node->obj;
            [ delete node ]
            break;
        }
    }
    return obj;
}
```

EPOCH-BASED RECLAMATION (EBR)

- Uses a ***global*** epoch counter (aka "era" in other algorithms)
- As part of per-thread state, each thread keeps a ***reservation***
- Many variations of EBR exist, which differ on how to increment the epoch counter (conditionally vs. unconditionally) and when to trigger memory reclamation
  - For the original EBR only 3 distinct epoch values are needed
- As an example, consider a variant with unconditional epoch increments presented in [PPoPP '18]

global_epoch = 2

reservations:

Thread 1	[epoch = 1]
Thread 2	[epoch = ∞]
Thread 3	[epoch = 2]
Thread 4	[epoch = ∞]

EPOCH-BASED RECLAMATION (EBR)

- Each data structure operation is wrapped
  - When ***beginning***, a thread records the current global epoch value to its reservation
  - When ***ending***, the thread resets its reservation

EPOCH-BASED RECLAMATION (EBR)

- Each data structure operation is wrapped
  - When ***beginning***, a thread records the current global epoch value to its reservation
  - When ***ending***, the thread resets its reservation

```
PUSH_EBR(Object* obj) {
    begin_op();
    PUSH(obj);
    end_op();
}
```

```
Object* POP_EBR() {
    begin_op();
    Object* obj = POP();
    end_op();
    return obj;
}
```

EPOCH-BASED RECLAMATION (EBR)

- Each data structure operation is wrapped
  - When ***beginning***, a thread records the current global epoch value to its reservation
  - When ***ending***, the thread resets its reservation

global_epoch = 2

```
begin_op() {
    reservations[TID] = global_epoch;
}
```

[epoch = ∞] → [epoch = 2]

EPOCH-BASED RECLAMATION (EBR)

- Each data structure operation is wrapped
  - When ***beginning***, a thread records the current global epoch value to its reservation
  - When ***ending***, the thread resets its reservation

global_epoch = 2

```
begin_op() {
    reservations[TID] = global_epoch;
}
```

[epoch = ∞] → [epoch = 2]

```
end_op() {
    reservations[TID] = ∞;
}
```

[epoch = 2] → [epoch = ∞]

EPOCH-BASED RECLAMATION (EBR)

- When deleting, postpone the actual deallocation by ***retiring*** a memory block
  - Store the global epoch counter at the moment of retiring ("retire epoch") and place the retired block to a thread-local list
  - Periodically increment the global epoch counter when retiring
  - Periodically scan previously retired blocks from the thread-local list and deallocate those for which the epoch at the moment of retirement is past all reservation values across ***all*** threads

global_epoch = 2

Thread 3's list

[retire=2] → [retire=2] → [retire=1] → [retire=0]

reservations:

Thread 1	[epoch = 1]
Thread 2	[epoch = ∞]
Thread 3	[epoch = 2]
Thread 4	[epoch = ∞]

EPOCH-BASED RECLAMATION (EBR)

- When deleting, postpone the actual deallocation by ***retiring*** a memory block
  - Store the global epoch counter at the moment of retiring (“retire epoch”) and place the retired block to a thread-local list
  - Periodically increment the global epoch counter when retiring
  - Periodically scan previously retired blocks from the thread-local list and deallocate those for which the epoch at the moment of retirement is past all reservation values across ***all*** threads

global_epoch = 2

Thread 3’s list [retire=2] → [retire=2] → [retire=1] → [retire=0]

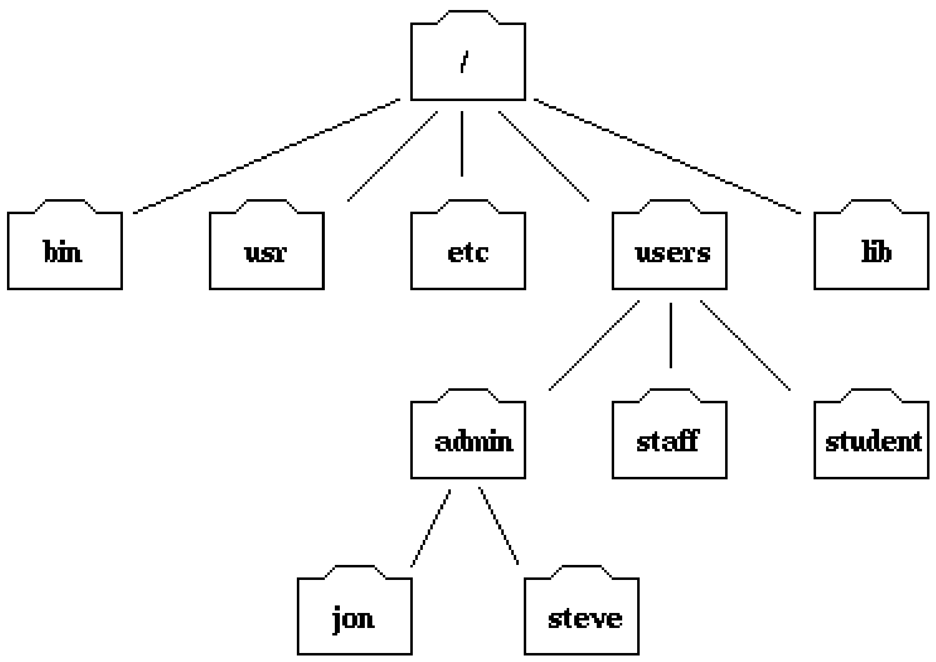
can be deleted

reservations:

Thread 1	[epoch = 1]
Thread 2	[epoch = ∞]
Thread 3	[epoch = 2]
Thread 4	[epoch = ∞]

Directories and filenames

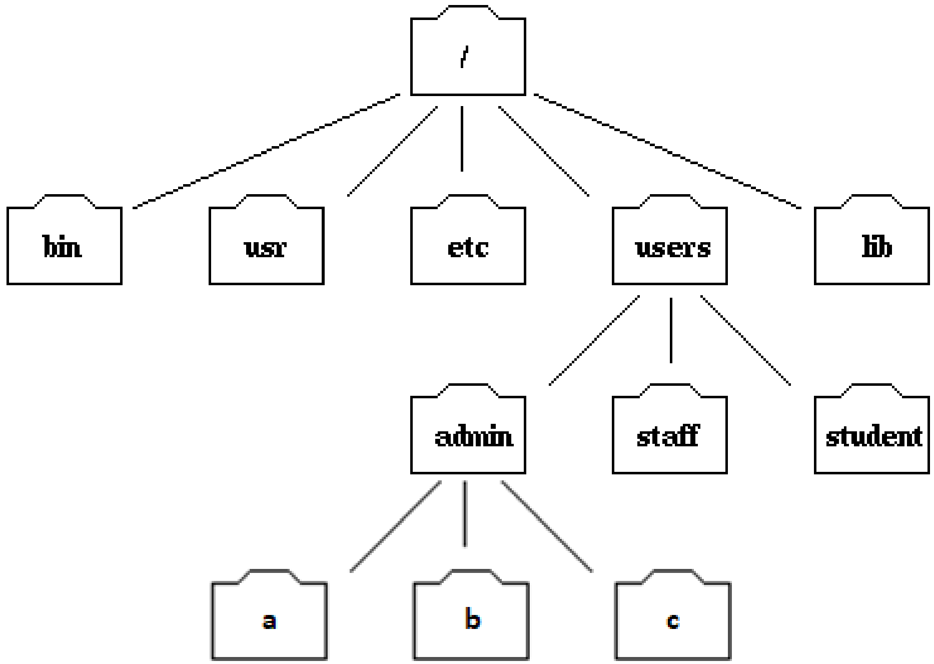
- Directory = list of files/directories
  - A directory is just a special file
  - Stored as a list of filenames + inode numbers



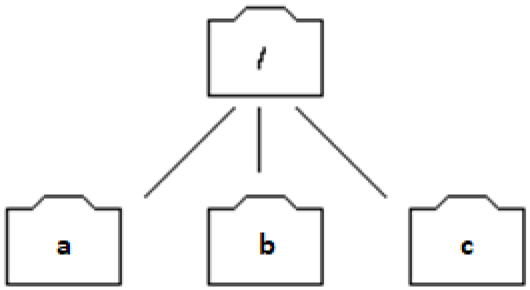
Mounting

- Attaching a directory into a directory tree
- Example: mount directory 2 into /users/admin

Directory 1



Directory 2



Question

Q: What is <u>not</u> contained within the inode metadata?

A. Size of file

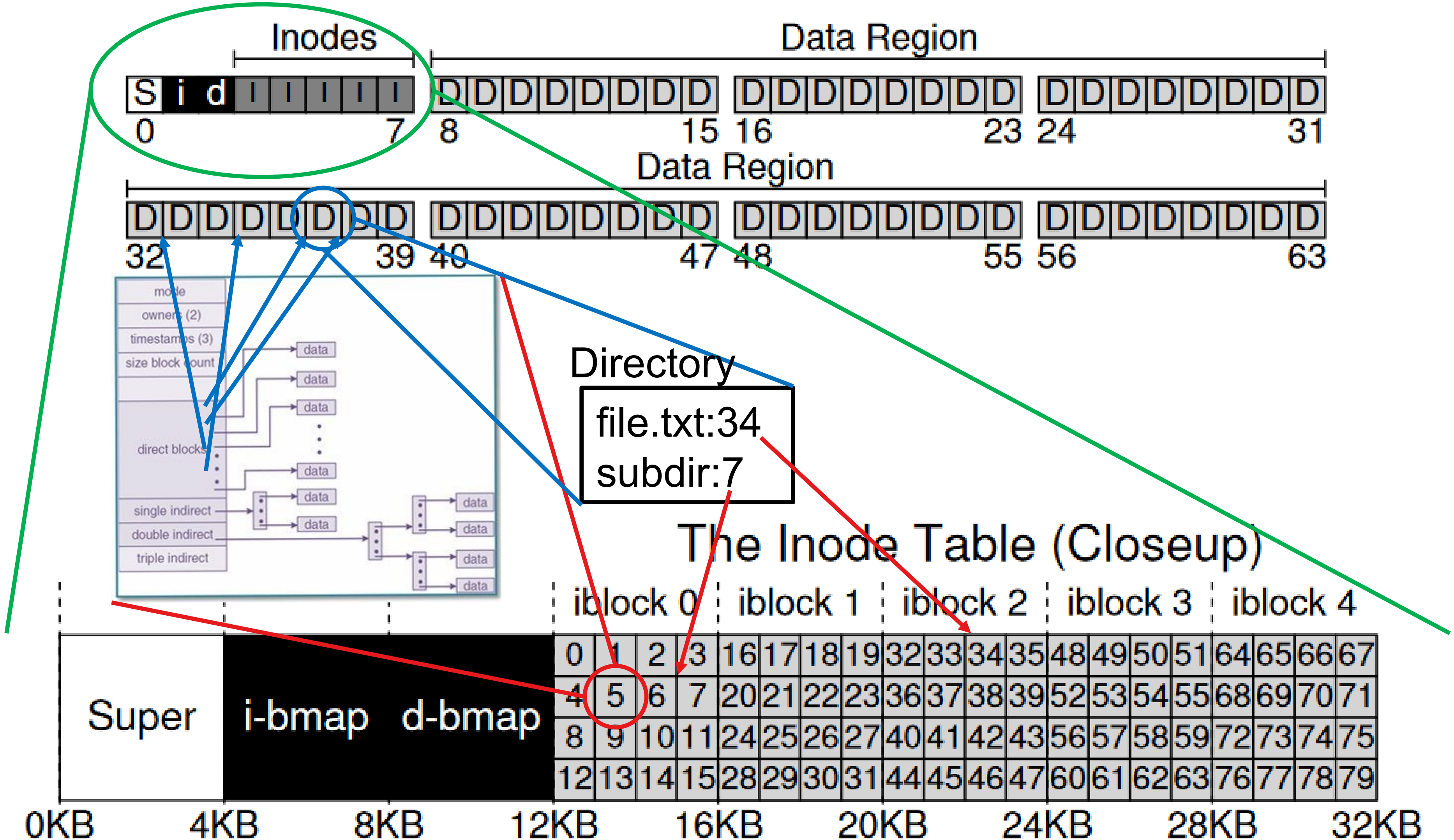
B. Filename

C. File owner

D. Timestamp of last modification

E. Access permission bits

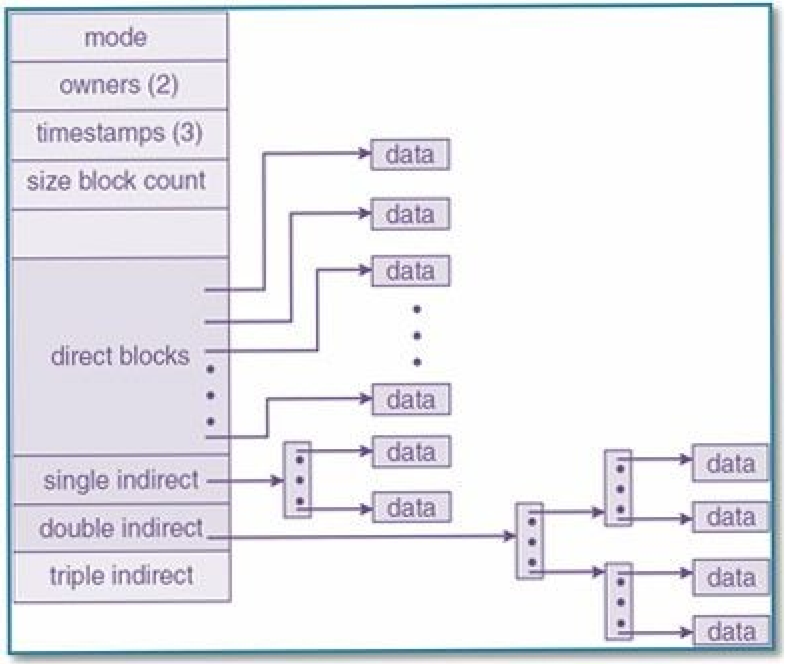
Example disk layout



Opening/reading a file

- int fd = open("/home/ruslan/file.txt", O_RDWR);
- char buffer[1024];
- read(fd, buffer, 1024);

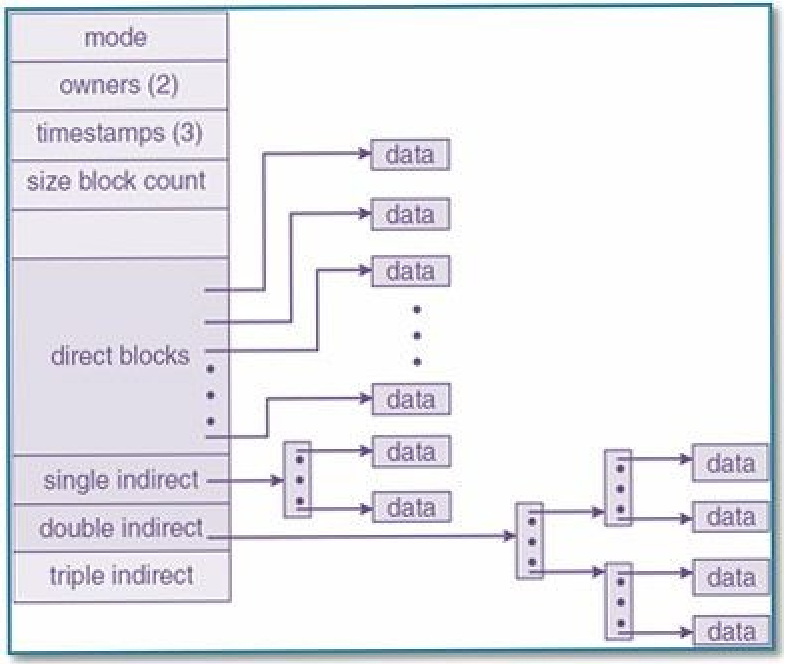
1. Read inode, corresponding to root directory / (fixed, system inode)
2. Read blocks for contents
3. Find inode # for "home"
4. Read inode for "home"
5. Read blocks for contents
6. Find inode # for "ruslan"
7. Read inode for "ruslan"



Opening/reading a file

- int fd = open("/home/ruslan/file.txt", O_RDWR);
- char buffer[1024];
- read(fd, buffer, 1024);

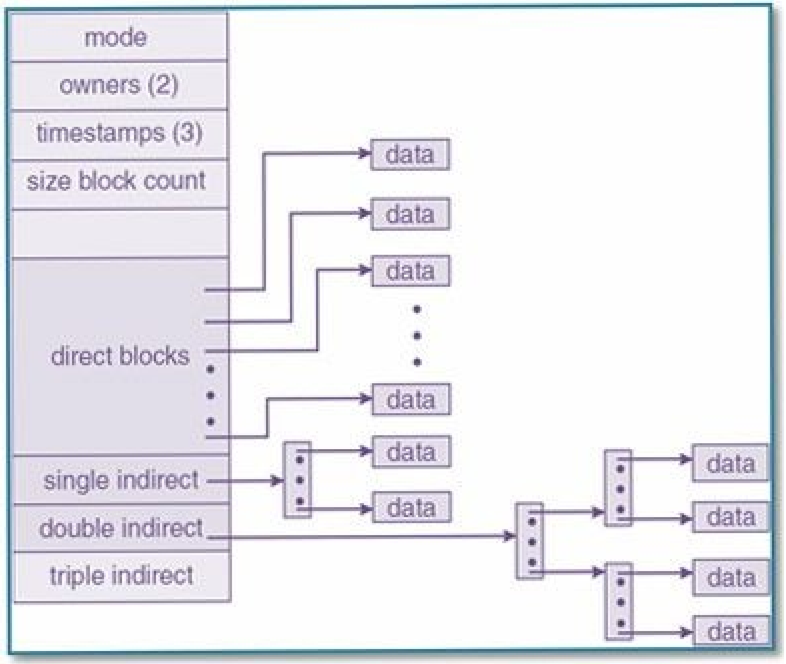
8. Read blocks for contents
9. Find inode # for "file.txt"
10. Read inode for "file.txt"
11. Check if user is owner
12. Check read and write bits
13. Allocate fd slot in fd table
14. Update fd table



Opening/reading a file

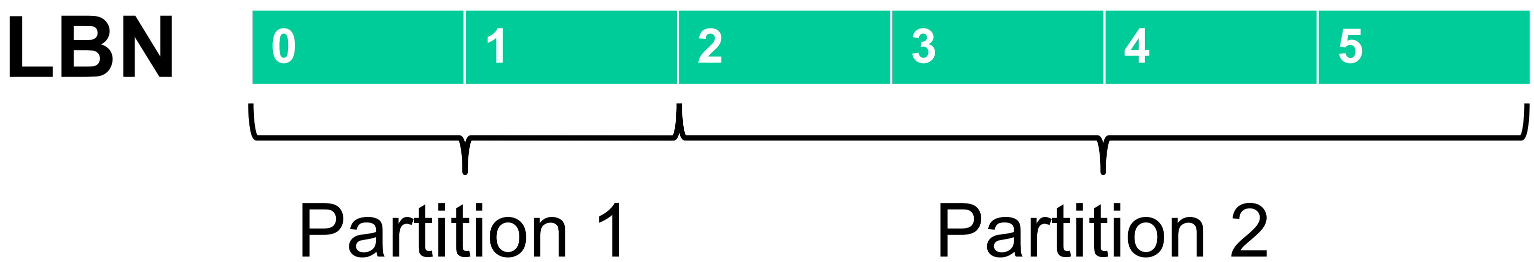
- int fd = open("/home/ruslan/file.txt", O_RDWR);
- char buffer[1024];
- read(fd, buffer, 1024);

15. Lookup inode # in fd table, indexed by fd
16. Read corresponding inode
17. Get first direct block #
18. Read first block into buffer
19. Get second direct block #
20. Read second block into buffer



Partitions

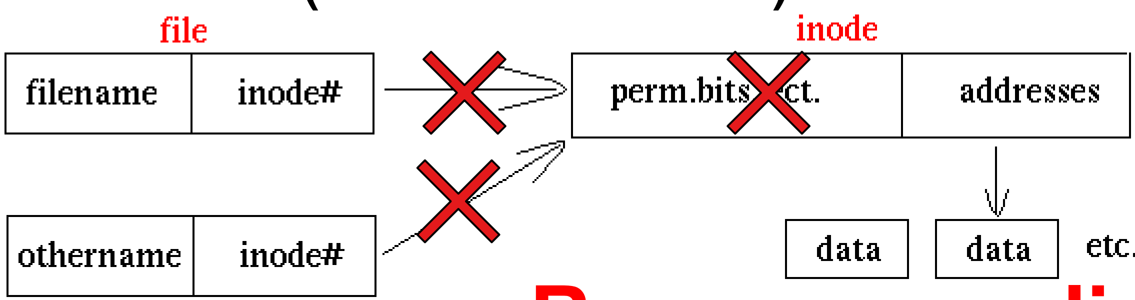
- Splitting storage into ranges of blocks



- Contiguous regions
- Fixed sizes (difficult to resize)
- Stored in a partition table
  - Master Boot Record (MBR) if you use legacy BIOS
  - GUID Partition Table (GPT) if you use UEFI

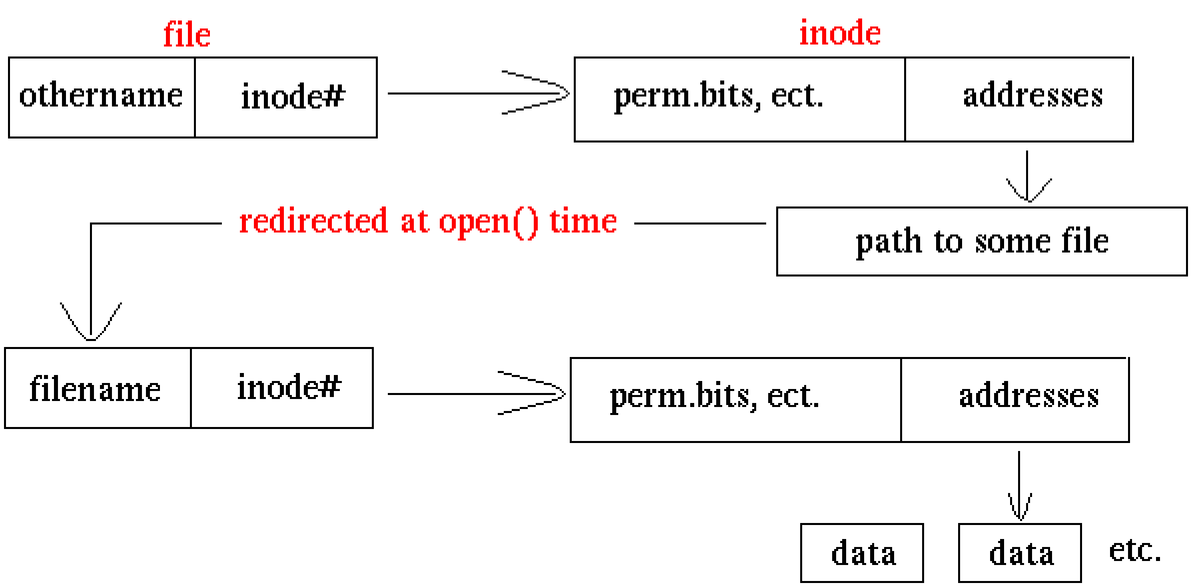
Hard links vs soft links

- Hard link (created via ln)



Remove = unlink

- Soft link (created via ln –s)



= Shortcut