Georgia Tech

Lecture 8: Buffer Management (Part 2)

CREATING THE NEXT®

Administrivia

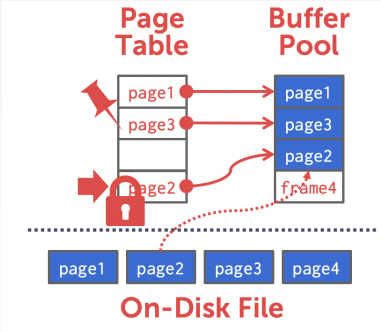
- We are delaying the project proposal to Sep 29 (Wednesday).
- We are delaying the mid-term exam to Oct 18 (Monday).
- We will cover lectures in the chapters 0 and 1 (into + storage management).
- You should ask questions about the **exercise sheet** on Piazza.

Today's Agenda

Recap

Buffer Pool Meta-Data

- The **page table** keeps track of pages that are currently in memory.
- Also maintains additional meta-data per page:
  - **Dirty Flag**
  - **Pin/Reference Counter**



Buffer Replacement Policies

- When the DBMS needs to free up a frame to make room for a new page, it must decide which page to evict from the buffer pool.
- Policies:
  - FIFO
  - LFU
  - LRU
  - CLOCK
  - LRU-k
  - 2Q

Buffer Manager Implementation

Buffer Manager Interface

Basic interface:

1. FIX (uint64_t page_id, bool is_shared)
2. UNFIX (uint64_t page_id, bool is_dirty)

Pages can only be accessed (or modified) when they are **fixed** in the buffer pool.

Segments

- Each table is organized a collection of **segments**.
- Each segments must be written into a separate file named after than segment's id

```
auto file_handle = File::open_file(std::to_string(segment_id).c_str(), File::WRITE);
file_handle->read_block(start, page_size_, pool_[frame_id]->data.data());
```

Segments

- Page id is split into **segment id** (16 bits) and **segment page id** (48 bits)
- Page id = **segment id** | **segment page id**
- We have provided helper functions to get this information

```
/// Returns the segment id for a given page id which is contained in the 16
/// most significant bits of the page id.
    static constexpr uint16_t get_segment_id(uint64_t page_id) {
    return page_id >> 48;
}

/// Returns the page id within its segment for a given page id. This
/// corresponds to the 48 least significant bits of the page id.
static constexpr uint64_t get_segment_page_id(uint64_t page_id) {
    return page_id & ((1ull << 48) - 1);
}
```

Bit Manipulation

int a = 33333, b = -77777; // 4 bytes

Expression	Representation	Value
a	00000000 00000000 10000010 00110101	33333
b	11111111 11111110 11010000 00101111	-77777
a & b	00000000 00000000 10000000 00100101	32805
a $\oplus$ b	11111111 11111110 01010010 00011010	-110054
a | b	11111111 11111110 11010010 00111111	-77249
(a | b)	00000000 00000001 00101101 11000000	77248
a & b	00000000 00000001 00101101 11000000	77248

Bit Manipulation

- If you want the k most significant bits of a value, then right shift the value by k
- Example: 1001 1100 » 4 = 0000 1001
- If you want the k least significant bits of a value, then apply a bit mask ((1ull « k) - 1)
- Example: 1 « 4 = 0001 0000; (1 « 4) - 1 = 0000 1111
- (1001 1100) & (0000 1111) = 0000 1100
- Reference

Bit Manipulation

Print an integer as a sequence of bits

```
include <limits.h>
include <stdio.h>

void bit_print(uint32_t a){
   int   i;
   int   n = sizeof(int) * CHAR_BIT;   /* number of bits in a byte (8) */
   int   mask = 1 << (n - 1);          /* mask = 100...0 */

   for (i = 1; i <= n; ++i) {
      putchar(((a & mask) == 0) ? '0' : '1');
      a <<= 1; // shifting left
      if (i % CHAR_BIT == 0 && i < n)
         putchar(' ');
   }
}
```

Bit Manipulation

Packing a set of bytes into an integer

```
include <limits.h>

/// Pack 4 characters into a 32-bit integer
uint32_t pack(char a, char b, char c, char d){
   uint32_t  p = a;     /* p will be packed with a, b, c, d */

   p = (p << CHAR_BIT) | b;
   p = (p << CHAR_BIT) | c;
   p = (p << CHAR_BIT) | d;
   return p;
}
```

Bit Manipulation

Unpacking a set of bytes from an integer

```
include <limits.h>

/// Unpack a byte from a 32-bit integer
char unpack(int p, int k){ /* k = 0, 1, 2, or 3 */
   int        n = k * CHAR_BIT;         /* n = 0, 8, 16, or 24 */
   unsigned   mask = ((1<<CHAR_BIT)-1); /* low-order byte */

   mask <<= n;
   return ((p & mask) >> n);
}
```

Thread Safety

Threads

- A **thread** of execution is a sequence of instructions that can be executed concurrently with other such sequences in **multi-threading** environments, while sharing a same **virtual address space**
- An initialized thread object represents an **active** thread of execution
- Such a thread object has a unique **thread id**
- One thread may wait for another thread to completes its execution
- This is known as **joining**

Threads

```
include <iostream>
include <utility>
include <thread>
include <chrono>

void foo(std::string msg){
    std::cout << "thread says: " << msg;
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

int main(){
    std::thread t1(foo, ``t1'');
    std::thread::id t1_id = t1.get_id();

    std::thread t2(foo, ``t2'');
    std::thread::id t2_id = t2.get_id();

    ...
```

Threads

```
int main(){
    ...

    std::cout << "t1's id: " << t1_id << '\n';
    std::cout << "t2's id: " << t2_id << '\n';

    t1.join();
    t2.join();
}
```

Thread Safety

- A piece of code is **thread-safe** if it functions correctly during simultaneous execution by multiple threads.
- In particular, it must satisfy the need for multiple threads to access the same shared data (**shared access**), and
- the need for a shared piece of data to be accessed by only one thread at any given time (**exclusive access**)

Thread Safety

- There are a few ways to achieve thread safety:
  - Atomic operations
  - Thread-local storage
  - Mutual exclusion

Atomic operations

- Shared data are accessed by using atomic operations which **cannot be interrupted** by other threads.
- This usually requires using **special assembly instructions**, which might be available in a runtime library.
- Since the operations are atomic, the shared data are always kept in a valid state, no matter how many other threads access it.
- Atomic operations form the basis of many **thread synchronization** mechanisms.
- C++: std::atomic

Example: American Idol App

We want to keep track of votes for each participant

```
int vote_counter = 0;

void vote (int number_of_votes) {
    for (int i=0; i<number_of_votes; ++i)  ++vote_counter;
}

int main (){
  std::vector<std::thread> threads;
  std::cout << "spawn 10 users...\n";
  for (int i=1; i<=10; ++i)
    threads.push_back(std::thread(vote, 20));

  std::cout << "joining all threads...\n";
  for (auto& th : threads) th.join();
  std::cout << "vote_counter: " << vote_counter << '\n';
  return 0;
```

Example: American Idol App

We want to keep track of votes for each participant

```
include <atomic>

std::atomic<int> vote_counter(0); // Using atomic

int main (){
  ...

  std::cout << "vote_counter: " << vote_counter << '\n';
  return 0;
}
```

Atomic operations

- Modern CPUs have direct support for atomic integer operations
- `LOCK` prefix in x86 ISA
- Example: lock incq 0x29a0(%rip)
- RIP addressing is **Relative** to 64-bit **Instruction Pointer** register
- std::atomic is a `portable` interface to those intructions
- Example: In aarch64 ISA, `LDADD` would be used instead

Thread-Local Storage

- Variables are localized so that each thread has its **own private copy**
- These variables retain their values across function and other code boundaries, and are thread-safe since they are local to each thread
- C++: thread_local

Example: American Idol App

We want to keep track of votes for each participant

```
include <atomic>

thread_local vote_counter = 0;

int main (){
  ...

  std::cout << "vote_counter: " << vote_counter << '\n';
  return 0;
}
```

- What will happen in this case?

Mutual exclusion

- Access to shared data is serialized using mechanisms that ensure **only one thread** reads or writes the shared data at any time.
- Great care is required if a piece of code accesses multiple shared pieces of data – problems include **race conditions**, **deadlocks**, **livelocks**, **starvation**, and various other ills enumerated in an OS textbook.
- Mutual exclusion is accomplished using **latches**
- C++: std::mutex

Example: American Idol App

We want to keep track of votes for each participant

```
include <mutex>

std::mutex vote_latch;
int vote_counter = 0;

void vote (int number_of_votes) {
        vote_latch.lock();
    for (int i=0; i<number_of_votes; ++i)  ++vote_counter;
        vote_latch.unlock();
}

int main (){
  ...

  std::cout << "vote_counter: " << vote_counter << '\n';
  return 0;
```

Mutual exclusion

- `std::mutex` is a more general method than `std::atomic`
- Can be used to make a **sequence of instructions** atomic
- But, slower than `std::atomic` because `std::mutex` makes futex system call in Linux
- Way slower than the userspace assembly instructions emitted by `std::atomic`

Lock Guard

- `lock_guard` is a mutex wrapper that provides a convenient **RAII-style mechanism** for owning a mutex for the duration of a scoped block.
- When a lock_guard object is created, it attempts to take ownership of the mutex it is given.
- When control leaves the scope in which the lock_guard object was created, the lock_guard is destructed and the mutex is released.

Example: American Idol App

We want to keep track of votes for each participant

```
include <mutex>

std::mutex vote_latch;
int vote_counter = 0;

void vote (int number_of_votes) {
    std::lock_guard<std::mutex> grab_latch(vote_latch);
    for (int i=0; i<number_of_votes; ++i)  ++vote_counter;
}

int main (){
  ...

  std::cout << "vote_counter: " << vote_counter << '\n';
  return 0;
}
```

Shared Mutex

- Shared mutexes are especially useful when shared data can be safely read by any number of threads simultaneously, but
- a thread may only write the same data when no other thread is reading or writing at the same time.
- The **shared_mutex** class is a synchronization primitive that can be used to protect shared data from being simultaneously accessed by multiple threads.
- In contrast to a regular `mutex` which facilitate exclusive access, a shared_mutex has two levels of access:
  - shared - several threads can share ownership of the same mutex
  - exclusive - only one thread can own the mutex

Shared Mutex

- If one thread has acquired the exclusive lock (through `lock`, `try_lock`), no other threads can acquire the lock (including the shared).
- If one thread has acquired the shared lock (through `lock_shared`, `try_lock_shared`), no other thread can acquire the exclusive lock, but can acquire the shared lock.
- Only when the exclusive lock has not been acquired by any thread, the shared lock can be acquired by multiple threads.
- Within one thread, only one lock (shared or exclusive) can be acquired at a given point in time.
  - **shared** - several threads can share ownership of the same mutex
  - **exclusive** - only one thread can own the mutex

Buffer Manager Implementation

- Must be thread-safe!
- Use `std::mutex` and `std::shared_mutex`
- Naiïve solution: Synchronize all accesses with a single latch
- Must be more efficient
  - Hold latches as short as possible
  - Do not hold latches while doing I/O operations
  - Distinguish between **shared** and **exclusive** requests

Buffer Manager Implementation

Synchronize accesses to segment

```
void BufferManager::read_frame(uint64_t frame_id) {

    std::lock_guard<std::mutex> file_guard(file_use_mutex_);
    ...

}
```

Buffer Manager Implementation

Write lock_frame and unlock_frame functions

```
void BufferManager::lock_frame(uint64_t frame_id, bool exclusive) {
  assert(frame_id != INVALID_FRAME_ID);
  assert(*use_counters_[frame_id] >= 0);

  if (exclusive == false) {
    lock_table_[frame_id]->lock_shared();
    pool_[frame_id]->exclusive = false;
    use_counters_[frame_id]->fetch_add(1);
  }
  else {
    lock_table_[frame_id]->lock();
    pool_[frame_id]->exclusive = true;
    pool_[frame_id]->exclusive_thread_id = std::this_thread::get_id();
    use_counters_[frame_id]->fetch_add(1);
  }
}
```

Buffer Manager Implementation

Write **copy constructor** and **copy assignment operator** for BufferFrame.

```
BufferFrame::BufferFrame(const BufferFrame& other)
    : page_id(other.page_id),
      frame_id(other.frame_id),
      data(other.data),
      dirty(other.dirty),
      exclusive(other.exclusive) {}

BufferFrame& BufferFrame::operator=(BufferFrame other) {
  std::swap(this->page_id, other.page_id);
  std::swap(this->frame_id, other.frame_id);
  std::swap(this->data, other.data);
  std::swap(this->dirty, other.dirty);
  std::swap(this->exclusive, other.exclusive);
  return *this;
}
```

Buffer Manager Implementation

- Reference counting (`use_counters_`) for eviction
- Fixing a page
  - Check if page alredy in buffer pool
  - If not found, find a free slot in the buffer pool
  - Lock the frame slot (exclusive mode)
  - Reset the frame slot's meta-data
  - Load data into the frame from disk
  - Unlock the frame slot (exclusive mode)
  - Lock the frame based on user's requested mode (exclusive or shared)

Buffer Manager Implementation

Fixing a page

```
BufferFrame& BufferManager::fix_page(uint64_t page_id, bool exclusive) {
  ...
  lock_frame(free_frame_id, true);
  // Reset meta-data
  pool_[free_frame_id]->page_id = page_id;
  pool_[free_frame_id]->dirty = false;
  read_frame(free_frame_id);
  // put in fifo queue
  {
    std::lock_guard<std::mutex> fifo_guard(fifo_mutex_);
    fifo_queue_.push_back(free_frame_id);
  }
  unlock_frame(free_frame_id);
  lock_frame(free_frame_id, exclusive);
  return *pool_[free_frame_id];
}
```

2Q Buffer Replacement Policy

2Q Policy

Maintain two queues (FIFO and LRU)

- Some pages are accessed only once (*e.g.*, sequential scan)
- Some pages are hot and accessed frequently
- Maintain separate lists for those pages
- **Scan resistant** policy

1. Maintain all pages in FIFO queue
2. When a page that is currently in FIFO is referenced again, upgrade it to the LRU queue
3. Prefer evicting pages from FIFO queue

Hot pages are in LRU, read-once pages in FIFO.

2Q Policy

Request: Fix(1, false)

FIFO Queue

-	-	-
-	-	-

LRU Queue

-	-	-
-	-	-

2Q Policy

Request: Fix(1, false) $\longrightarrow$ True

FIFO Queue

1	-	-
S	-	-

LRU Queue

-	-	-
-	-	-

2Q Policy

Request: Fix(2, true)

FIFO Queue

1	-	-
S	-	-

LRU Queue

-	-	-
-	-	-

2Q Policy

Request: Fix(2, true) $\longrightarrow$ True

FIFO Queue

1	2	-
S	X	-

LRU Queue

-	-	-
-	-	-

2Q Policy

Request: Fix(3, false)

FIFO Queue

1	2	-
S	X	-

LRU Queue

-	-	-
-	-	-

2Q Policy

Request: Fix(3, false) $\longrightarrow$ True

FIFO Queue

1	2	3
S	X	S

LRU Queue

-	-	-
-	-	-

2Q Policy

Request: Fix(4, false)

FIFO Queue

1	2	3
S	X	S

LRU Queue

-	-	-
-	-	-

2Q Policy

Request: Fix(4, false) $\longrightarrow$ False // (throw buffer_full_error{})

FIFO Queue

1	2	3
S	X	S

LRU Queue

-	-	-
-	-	-

2Q Policy

Request: Unfix(1, false)

FIFO Queue

1	2	3
S	X	S

LRU Queue

-	-	-
-	-	-

2Q Policy

Request: Unfix(1, false) $\longrightarrow$ True

FIFO Queue

-	2	3
-	X	S

LRU Queue

-	-	-
-	-	-

2Q Policy

Request: Fix(4, false)

FIFO Queue

-	2	3
-	X	S

LRU Queue

-	-	-
-	-	-

2Q Policy

Request: Fix(4, false)

FIFO Queue

2	3	-
X	S	-

LRU Queue

-	-	-
-	-	-

2Q Policy

Request: Fix(4, false) $\longrightarrow$ True

FIFO Queue

2	3	4
X	S	S

LRU Queue

-	-	-
-	-	-

2Q Policy

Request: Fix(4, false)

FIFO Queue

2	3	4
X	S	S

LRU Queue

-	-	-
-	-	-

2Q Policy

Request: Fix(4, false) $\longrightarrow$ True

FIFO Queue

2	3	-
X	S	-

LRU Queue

4	-	-
S	-	-

2Q Policy

Request: Unfix(2, true)

FIFO Queue

2	3	-
X	S	-

LRU Queue

4	-	-
S	-	-

2Q Policy

Request: Unfix(2, true) $\longrightarrow$ True

FIFO Queue

3	-	-
S	-	-

LRU Queue

4	-	-
S	-	-

Fix Page

```
BufferFrame& BufferManager::fix_page(uint64_t page_id, bool exclusive) {
        // first check if page is in lru queue: if found, return the frame
        // if not, check for page in fifo queue: if found, return the frame
        // if not, find a free slot
        //   - is the buffer full?
        //   - if it is not full, get the next available slot
        //   - if it is full, find a free slot in fifo queue
        //   -                find a free slot in lru queue
        //   - throw buffer_full error
        // found a free slot
        // lock frame in exclusive mode
        // set frame's meta-data
        // read frame from disk using frame's meta-data
        // add frame to fifo queue
        // unlock frame in exclusive mode
        // lock frame in user's requested mode
        // return the frame
```

Page in FIFO Queue

```
std::pair<bool, uint64_t> BufferManager::page_in_fifo_queue(uint64_t page_id) {
  {
    std::lock_guard<std::mutex> fifo_guard(fifo_mutex_);
    std::lock_guard<std::mutex> lru_guard(lru_mutex_);
    bool found_page = false;
    uint64_t page_frame_id = INVALID_FRAME_ID;
    for (size_t i = 0; i < fifo_queue_.size(); i++) {
      auto frame_id = fifo_queue_[i];
      if (pool_[frame_id]->page_id == page_id) {
        found_page = true;
        page_frame_id = frame_id;
        fifo_queue_.erase(fifo_queue_.begin() + i);
        lru_queue_.push_back(frame_id);
        break;
      }
    }
    return std::make_pair(found_page, page_frame_id);
  }
```

Conclusion

- Thread-safety is an important required with modern multi-core processors
- We maximize concurrency in the buffer manager by:
  - Holding latches as short as possible
  - Not holding latches while doing I/O operations
  - Distinguishing between shared and exclusive requests
- In the next lecture, we will learn about compression.