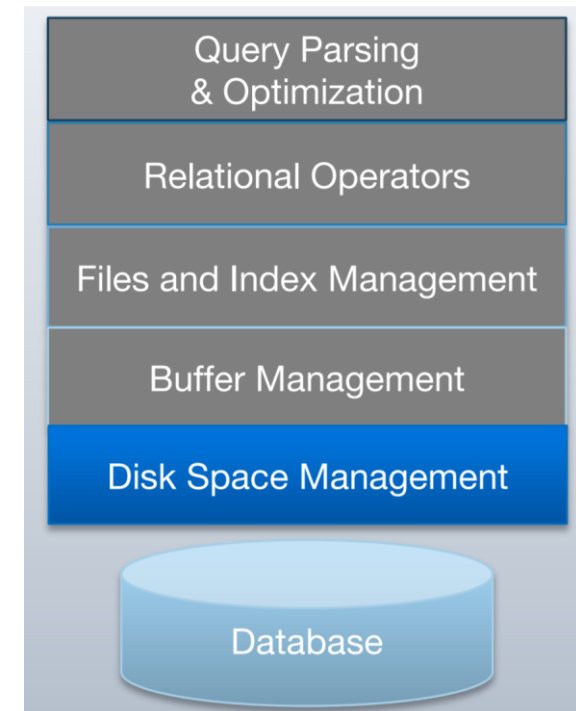


Project2 milestone 1

Disk Space Manager

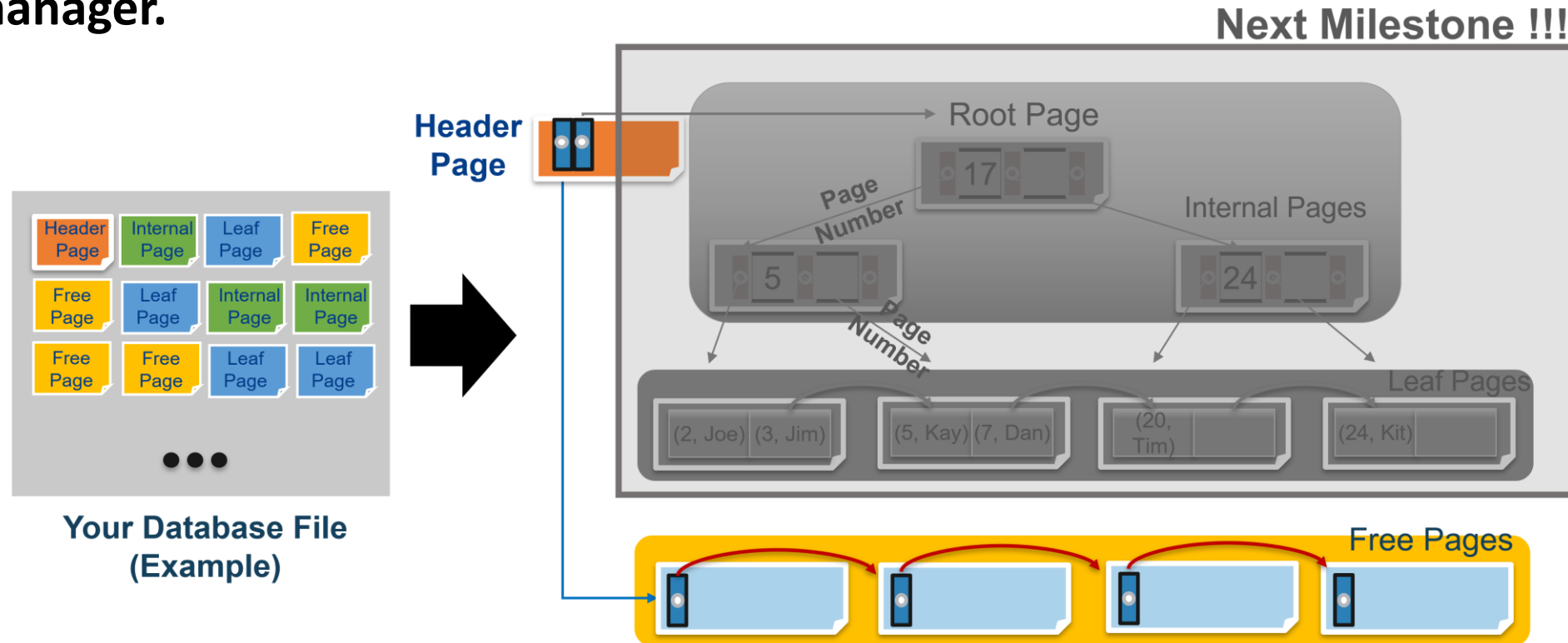
Recall Disk Space Manager

- When representing the DBMS in a layered architecture, the disk space manager is the lowest layer of DBMS.
- The disk space manager manages space on disk by:
 - mapping pages to locations on disk.
 - loading pages from disk to memory.
 - saving pages back to disk & ensuring writes.



A Final Picture of Your Disk-based B+tree

- Project2 is to implement disk-based b+tree based on your disk space manager.
- Your disk-based b+tree consists of **disk space manager** (milestone 1) and **index manager**.

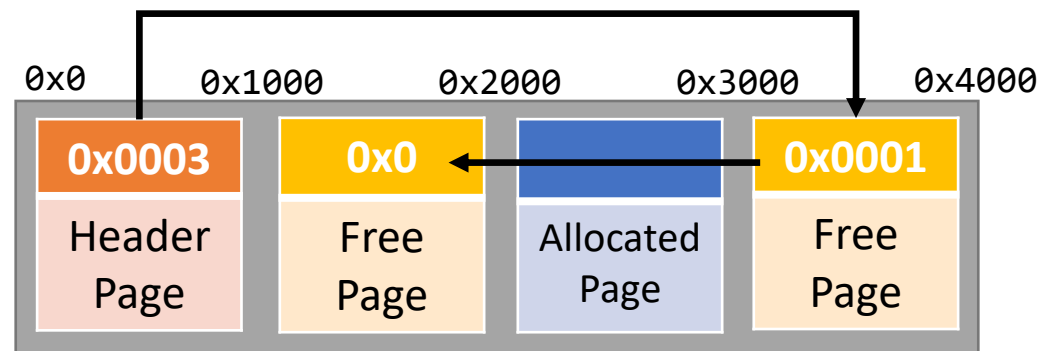


Layered Architecture

- A well-designed layered architecture can benefit a lot by isolating issues between levels.
 - Even if a new layer is added, many code changes may not be necessary.
 - The influence of code changes can be limited, so coding or testing can be carried out layer by layer.
- It means that it is important to be able to operate each layer in isolation and intact for carrying out the entire projects.
- First, you should implement the disk space manager that works well.

Overall Architecture of Milestone 1

- In milestone1, the database file consists of a set of 4KiB pages.
- Each page is roughly divided into three types:
 - Header Page
 - Free Page
 - Allocated Page



Database file example

Milestone Specification

- Implement **six important APIs** that should be exported to its upper layer :
open database file / allocate page / free page / read page / write page / close database file
- **All I/O must be performed in a unit of pages.**
- Validation of read, write, allocation, and free should be done using Google unittest.
 1. **int file_open_database_file (const char * pathname)**
 - Open the database file.
 - It opens an existing database file using 'pathname' or create a new file if absent.
 - If a new file needs to be created, the default file size should be **10 MiB**.
 - Then it returns the file descriptor of the opened database file.
 - All other 5 commands below should be handled after open data file.
 2. **uint64_t file_alloc_page (int fd);**
 - Allocate a page.
 - It returns a new page # from the free page list.
 - If the free page list is empty, then it should grow the database file and return a free page #.

Milestone Specification

- Implement **six important APIs** that should be exported to its upper layer :
open database file / allocate page / free page / read page / write page / close database file
- **All I/O must be performed in a unit of pages.**
- Validation of read, write, allocation, and free should be done using Google unittest.
 - 3. **void file_free_page (int fd, uint64_t page_number);**
 - Free a page.
 - It informs the disk space manager of returning the page with 'page_number' for freeing it to the free page list.
 - 4. **void file_read_page (int fd, uint64_t page_number, char * dest);**
 - Read a page.
 - It fetches the disk page corresponding to 'page_number' to the in-memory buffer (i.e., 'dest').
 - 5. **void file_write_page (int fd, uint64_t page_number, const char * src);**
 - Write a page.
 - It writes the in-memory page content in the buffer (i.e., 'src') to the disk page pointed by 'page_number'.

Milestone Specification

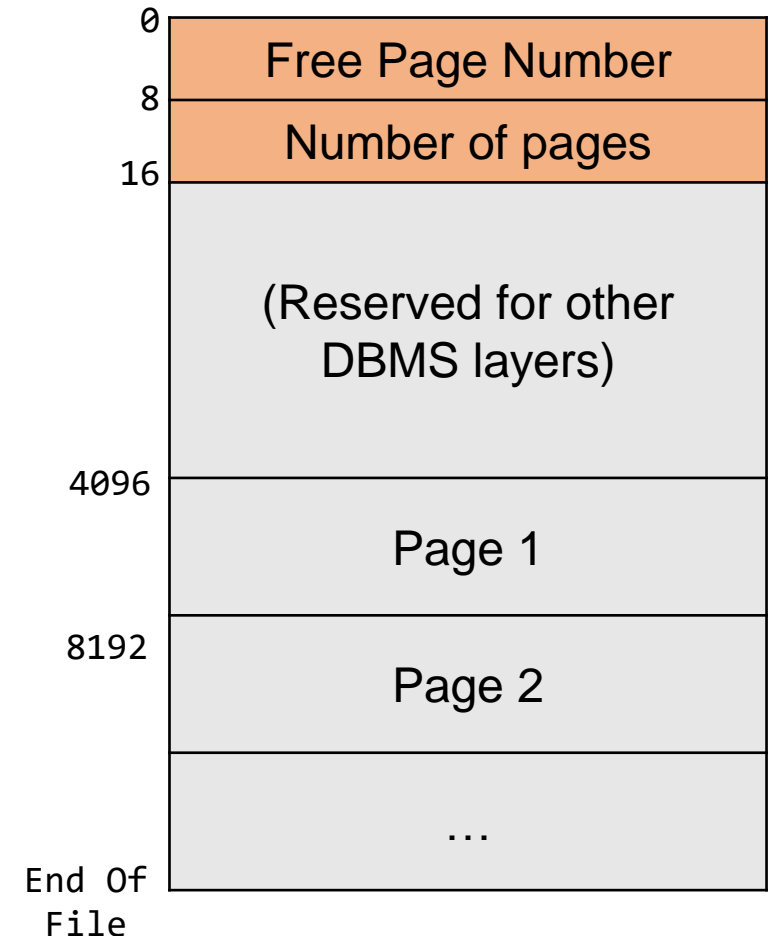
- Implement **six important APIs** that should be exported to its upper layer :
open database file / allocate page / free page / read page / write page / close database file
- **All I/O must be performed in a unit of pages.**
- Validation of read, write, allocation, and free should be done using Google unittest.

6. **void file_close_database_file();**

- Close the database file.
- This API doesn't receive a file descriptor as a parameter. So a means for referencing the descriptor of the opened file(i.e., global variable) is required.

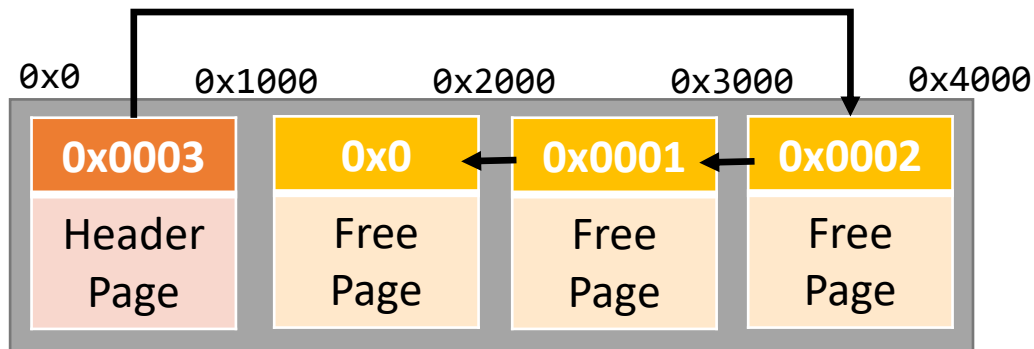
Header Page Format

- Header page is the **first page (offset 0-4095)** of a database file and contains important metadata to manage a list of free pages and file space.
- Free page number: byte range [0-7]
 - It points to the first free page (head of free page list)
 - 0, if there is no free page left.
- Number of pages: byte range [8-15]
 - It tells how many pages are paginated in this database file. (Count the header page itself as well.)
- Later, you need to use some fields in “reserved” space.



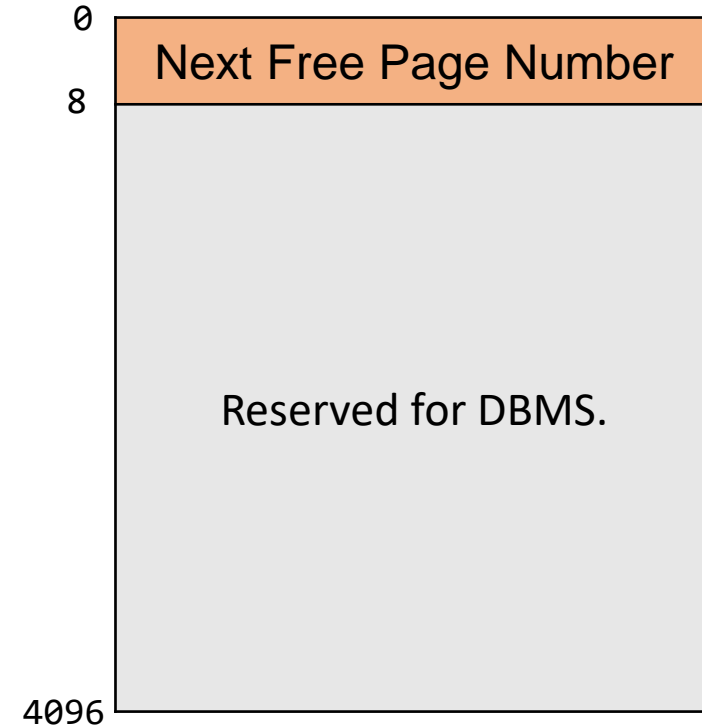
Free Page Format

- In the previous slide, the header page contains the position of the first free page in the list.
- Free pages are all linked, and page allocation is done by getting a free page from the free page list.
- **Next Free Page Number**: byte range [0-7]
 - points the next free page.
 - 0, if end of the free page list.



Database file example

Free Page Layout



Disk Space Manager APIs

–exported to its upper layer–

```
// file.h
#include <stdint.h>

typedef uint64_t pagenum_t;
struct page_t {
    // in-memory page structure
};

// Open existing database file or create one if not existed.
int file_open_database_file(const char* pathname);

// Allocate an on-disk page from the free page list
pagenum_t file_alloc_page(int fd);

// Free an on-disk page to the free page list
void file_free_page(int fd, pagenum_t pagenum);
```

Disk Space Manager APIs (cont.)

–exported to its upper layer–

```
// file.h

// Read an on-disk page into the in-memory page structure(dest)
void file_read_page(int fd, pagenum_t pagenum, page_t* dest);

// Write an in-memory page(src) to the on-disk page
void file_write_page(int fd, pagenum_t pagenum,
                    const page_t* src);

// Close the database file
void file_close_database_file();
```

Disk Space Manager APIs (cont.)

```
// file.c or file.cpp

// Open existing database file or create one if not existed.
int file_open_database_file(const char* pathname){
}

// Allocate an on-disk page from the free page list
pagenum_t file_alloc_page(int fd){
}

// Free an on-disk page to the free page list
void file_free_page(int fd, pagenum_t pagenum){
}
```

Disk Space Manager APIs (cont.)

```
// file.c or file.cpp

// Read an on-disk page into the in-memory page structure(dest)
void file_read_page(int fd, pagenum_t pagenum,
                    page_t* dest){
}

// Write an in-memory page(src) to the on-disk page
void file_write_page(int fd, pagenum_t pagenum,
                    const page_t* src){
}

// Close the database file
void file_close_database_file(){
}
```

Disk Space Manager APIs (cont.)

- You have to export and implement a set of APIs for managing a database file and synchronizing between in-memory pages and on-disk pages.
- There are other internal APIs you have to implement for testing and debugging purposes.
- **Once disk space manager's APIs are ready for use, there must be the upper layer component (B-tree in the first project) who can invoke these APIs.**
 - **If you violate this layered architecture principle, your project will be ruined and get *zero score* in other projects following the first one.**

Disk Space Manager APIs (cont.)

- When performing a write operation to disk, it is not immediately written to disk. This is because of the method used by the OS to reduce I/O which has a big negative impact on performance.
- However, in this project, the page on disk and the page in memory should be synchronized.
- To force any write to hit the disk , you need to call a function like *fsync()* or *sync()* after calling the *fwrite()* or *write()* function.
- Proper synchronization between disk and memory pages is also included in the scoring scope.

File Allocation Rule

- When new pages are allocated by calling `file_alloc_page()`, new pages must be allocated as much as the current DB size.
- This is because the operation of allocating a free page every time it is needed can cause performance degradation.
- Once you implemented all the required APIs, then you have to test the correctness by using Google unit test framework, which will be announced separately with more details on how to setup gtest with some concrete examples.
- The following is a link to the user guide of google unit test framework:
 - <https://google.github.io/googletest/>
- **Doubling the current file space is reasonable and recommended in this project.**

GoogleTest

- **GoogleTest** is Google's C++ testing and mocking framework.
- This tutorial is a quick-start package.
- As aforementioned, we will later announce a more detailed tutorial with more sophisticated and useful usages.
- Until then, you can get used to **GoogleTest** by using this sandbox.

GoogleTest

```
~ /code ➤ mkdir googletest
~ /code ➤ cd googletest
~ /c/googletest ➤ pwd
/home/jaechan/code/googletest
~ /c/googletest ➤
```

- Make a directory named 'googletest'
- Change directory to 'googletest'

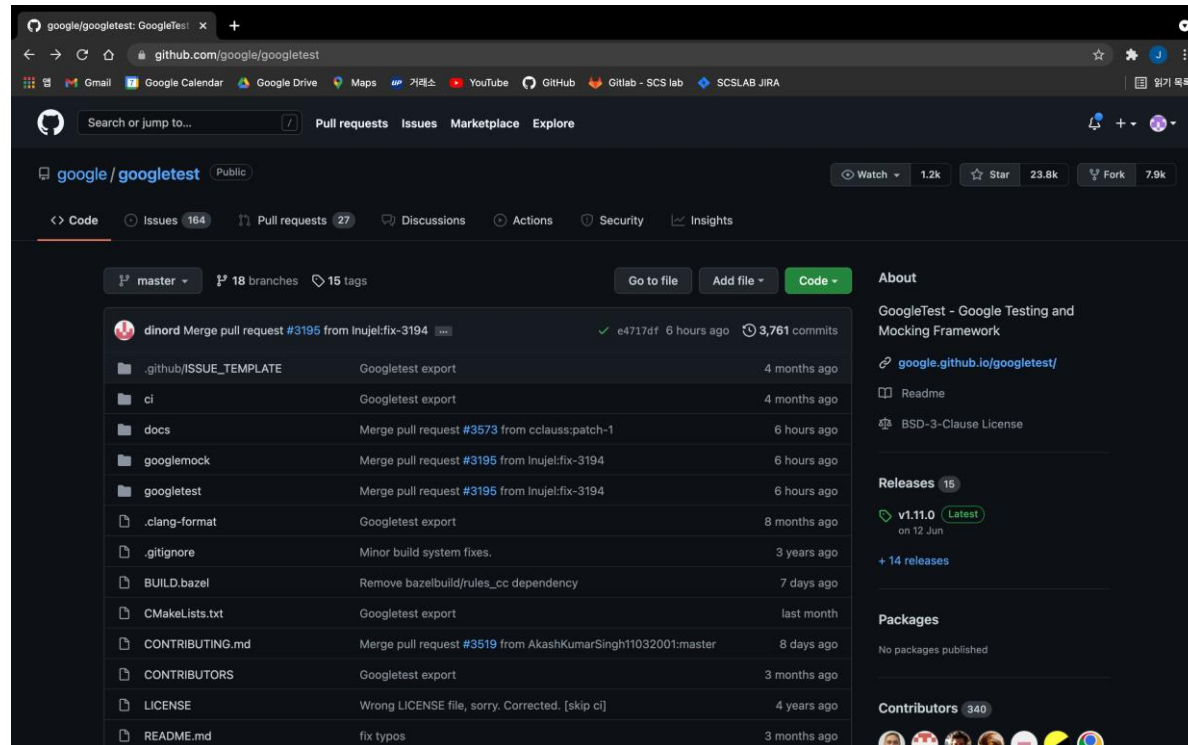
GoogleTest

```
~ /c/googletest touch CMakeLists.txt
~ /c/googletest touch basic_test.cc
~ /c/googletest ls
basic_test.cc CMakeLists.txt
~ /c/googletest
```

- Create a file named 'CMakeList.txt'
- Create a file named 'basic_test.cc'
- Make sure the files are within the directory that you've created (i.e., googletest)

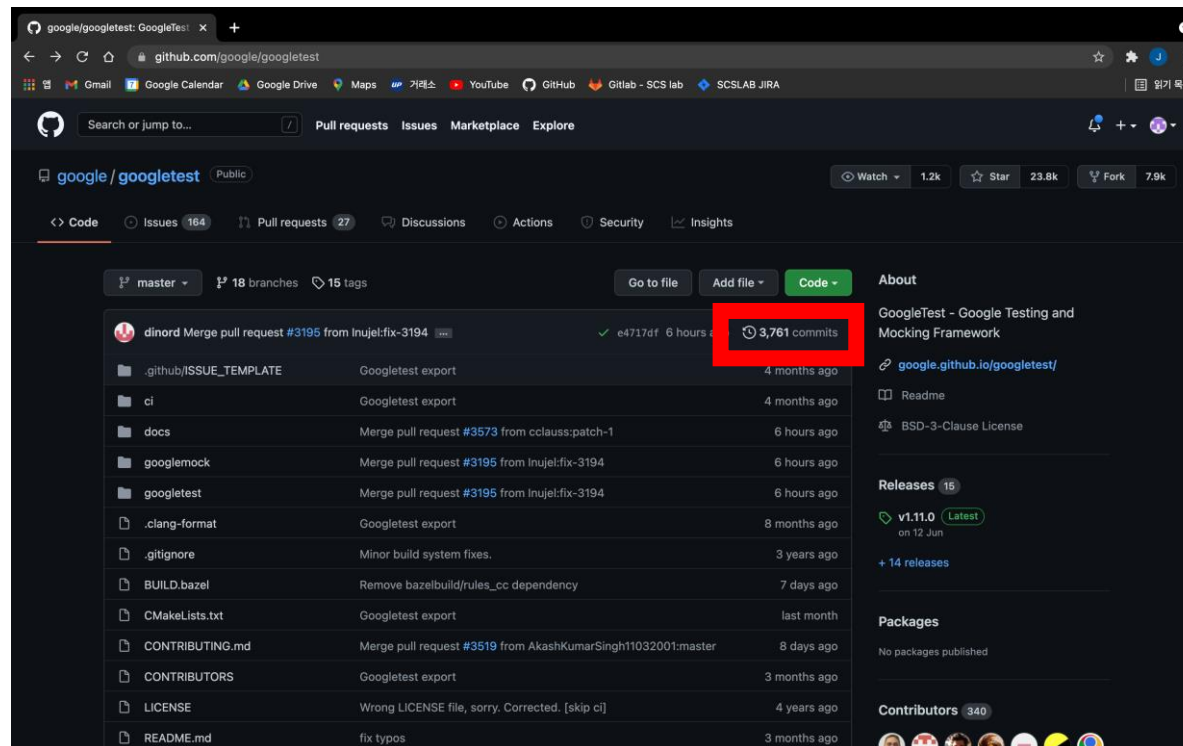
GoogleTest

- Go to the googletest github repository
- <https://github.com/google/googletest>



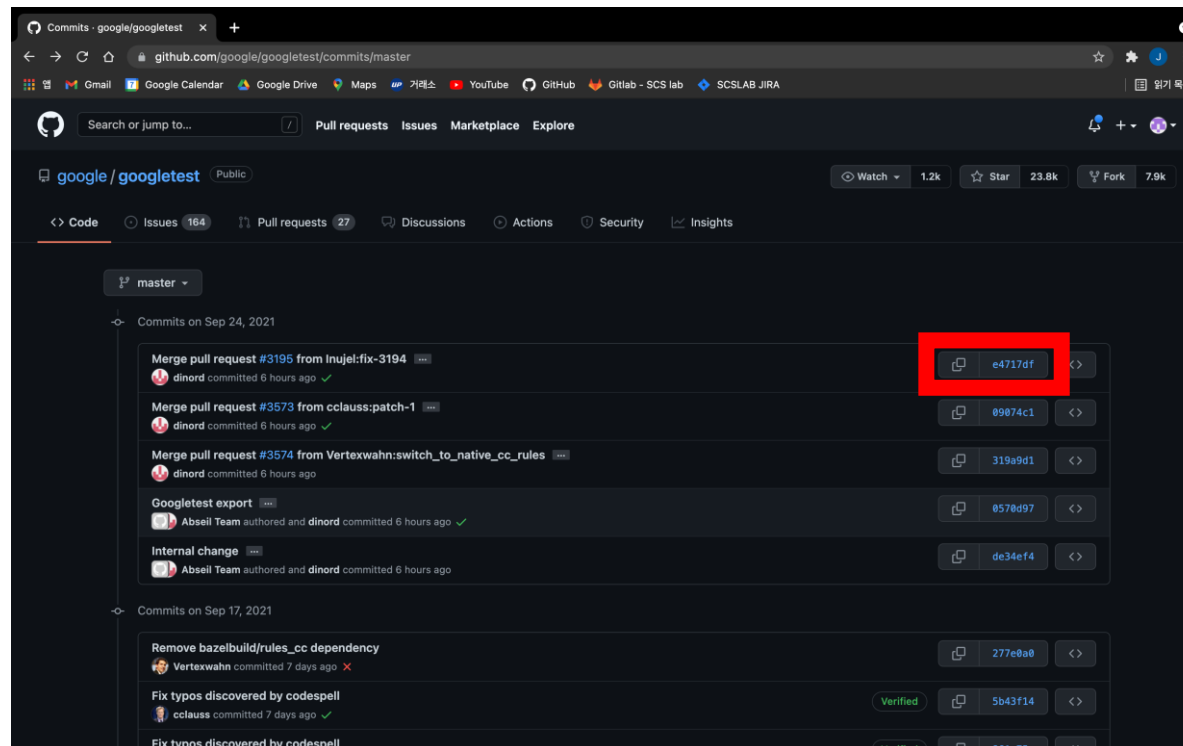
GoogleTest

- Press 'commits' as in the red box in the picture below



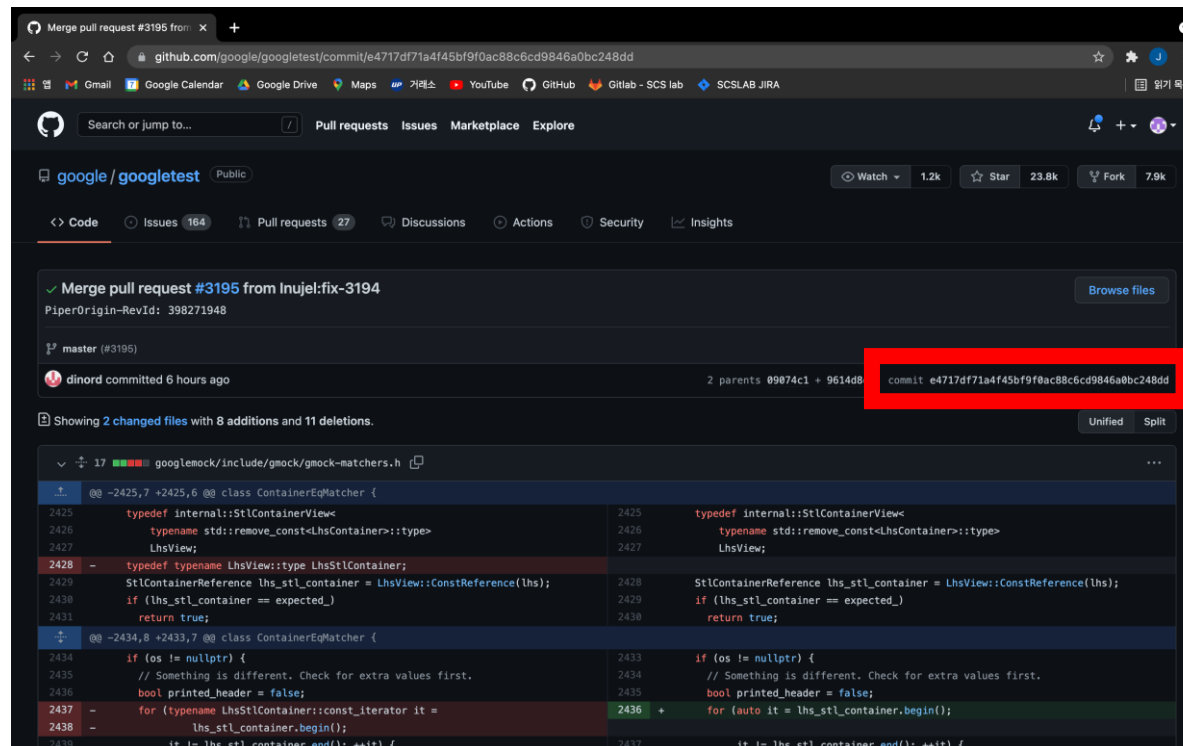
GoogleTest

- Press the latest commit as in the red box in the picture below



GoogleTest

- Memo the latest commit's hash somewhere
- We will use it in the following steps

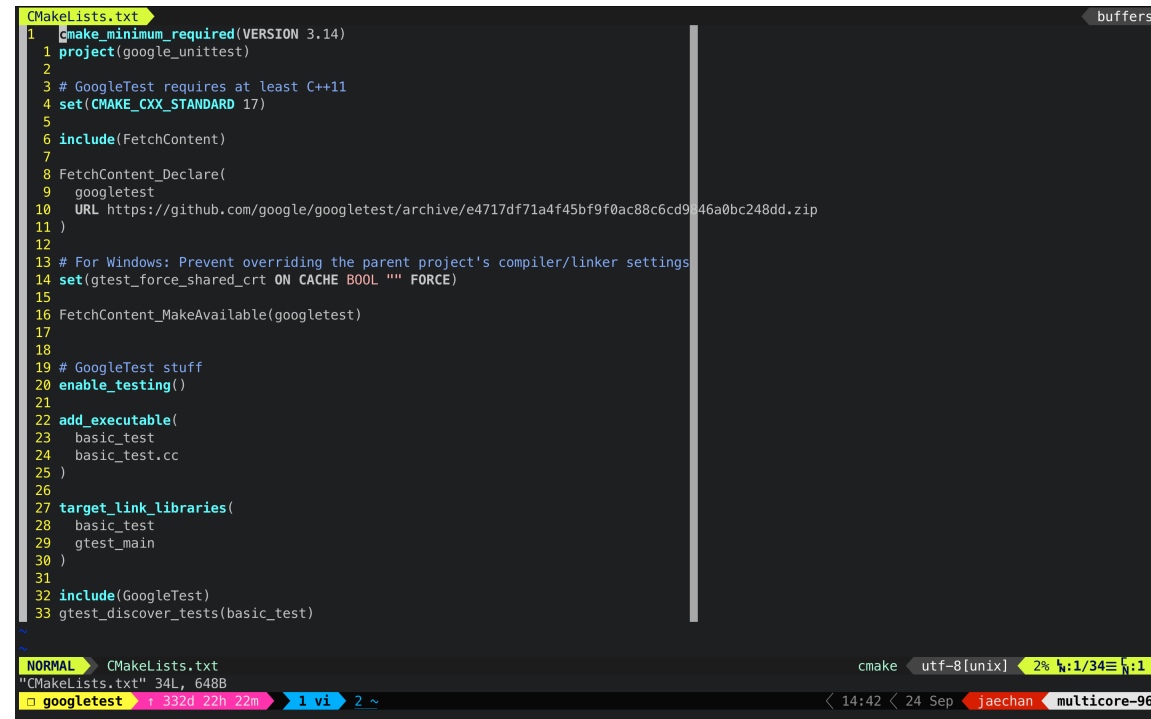


The screenshot shows a GitHub pull request #3195 from Inujel:fix-3194. The commit hash e471df71a4f45bf9f0ac88c6cd9846a0bc248dd is highlighted in a red box. The diff shows changes to the file gmock/include/gmock/gmock-matchers.h. The changes include adding a new typedef for LhsStlContainer, updating the LhsView typedef, and modifying the ContainerEqMatcher class to use the new typedef and a range-based for loop.

```
@@ -2425,7 +2425,6 @@ class ContainerEqMatcher {
2425     typedef internal::StlContainerView<
2426         typename std::remove_const<LhsContainer>::type>
2427         LhsView;
2428 -    typedef typename LhsView::type LhsStlContainer;
2429     StlContainerReference lhs_stl_container = LhsView::ConstReference(lhs);
2430     if (lhs_stl_container == expected_)
2431         return true;
@@ -2434,8 +2433,7 @@ class ContainerEqMatcher {
2434     if (os != nullptr) {
2435         // Something is different. Check for extra values first.
2436         bool printed_header = false;
2437 -        for (typename LhsStlContainer::const_iterator it =
2438 -             lhs_stl_container.begin();
2439             it != lhs_stl_container.end(); ++it) {
2433     if (os != nullptr) {
2434         // Something is different. Check for extra values first.
2435         bool printed_header = false;
2436 +        for (auto it = lhs_stl_container.begin();
2437             it != lhs_stl_container.end(); ++it) {
```


GoogleTest


- Write the following text within your CMakeLists.txt file
 - See the next page for a larger picture
- Remember to change the URL's hash to the latest hash you've fetched



```
CMakeLists.txt
1 cmake_minimum_required(VERSION 3.14)
2 project(google_unittest)
3
4 # GoogleTest requires at least C++11
5 set(CMAKE_CXX_STANDARD 17)
6
7 include(FetchContent)
8
9 FetchContent_Declare(
10   googletest
11   URL https://github.com/google/googletest/archive/e4717df71a4f45bf9f0ac88c6cd946a0bc248dd.zip
12 )
13 # For Windows: Prevent overriding the parent project's compiler/linker settings
14 set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
15
16 FetchContent_MakeAvailable(googletest)
17
18 # GoogleTest stuff
19 enable_testing()
20
21 add_executable(
22   basic_test
23   basic_test.cc
24 )
25
26 target_link_libraries(
27   basic_test
28   gtest_main
29 )
30
31 include(GoogleTest)
32 gtest_discover_tests(basic_test)
```

The screenshot shows a code editor with a dark theme. The file name 'CMakeLists.txt' is in the top left. The code is CMake configuration for GoogleTest. Line 10 contains a URL to a GoogleTest archive. The status bar at the bottom shows 'cmake', 'utf-8[unix]', '2%', '1/34', and '1 vi 2 ~'. There is also a timestamp '14:42' and a date '24 Sep'.

```
1 cmake_minimum_required(VERSION 3.14)
2 project(google_unittest)
3 # GoogleTest requires at least C++11
4 set(CMAKE_CXX_STANDARD 17)
5
6 include(FetchContent)
7
8 FetchContent_Declare(
9   googletest
10  URL https://github.com/google/googletest/archive/e4717df71a4f45bf9f0ac88c6cd9846a0bc248dd.zip
11 )
12
13 # For Windows: Prevent overriding the parent project's compiler/linker settings
14 set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
15
16 FetchContent_MakeAvailable(googletest)
17
18
19 # GoogleTest stuff
20 enable_testing()
21
22 add_executable(
23   basic_test
24   basic_test.cc
25 )
26
27 target_link_libraries(
28   basic_test
29   gtest_main
30 )
31
32 include(GoogleTest)
33 gtest_discover_tests(basic_test)
```



Put your hash here

NORMAL CMakeLists.txt

cmake utf-8[unix] 2% 1/34 1

"CMakeLists.txt" 34L, 648B

googletest 332d 22h 22m 1 vi 2 ~

< 14:42 < 24 Sep < jaechan < multicore-96

GoogleTest

```
basic_test.cc
1 #include <gtest/gtest.h>
2 #include <cmath>
3 #include <vector>
4 int Factorial(int n) {
5     if (n == 0) {
6         return 1;
7     }
8
9     int result = 1;
10    for (int i = 2; i <= n; i++) {
11        result *= i;
12    }
13    return result;
14 }
15
16 bool IsPrime(int n) {
17     if (n < 2) {
18         return false;
19     }
20
21     for (int i = 2; i <= sqrt(n); i++) {
22         if (n % i == 0) {
23             return false;
24         }
25     }
26     return true;
27 }
28
29
NORMAL basic_test.cc cp
```

- Open `basic_test.cc` and create the following two functions, *Factorial* and *IsPrime*
- Make sure that you include '*gtest/gtest.h*'

GoogleTest

```
30 // Tests factorial of 0
1 TEST(FactorialTest, HandlesZeroInput) {
2     // Tests if the value of Factorial(0) is EQ(equal) to 1
3     EXPECT_EQ(Factorial(0), 1);
4 }
5
6 // Tests factorial of positive numbers
7 TEST(FactorialTest, HandlesPositiveInput) {
8     EXPECT_EQ(Factorial(1), 1);
9     EXPECT_EQ(Factorial(2), 2);
10    EXPECT_EQ(Factorial(3), 6);
11    EXPECT_EQ(Factorial(8), 40320);
12 }
```

```
44 // Tests if 1 is a prime or not
1 TEST(PrimeTest, HandlesOneInput) {
2     // 1 is not a prime
3     EXPECT_FALSE(IsPrime(1));
4 }
5
6 // Tests if a given positive number is a prime or not
7 TEST(PrimeTest, HandlesPositiveInput) {
8     EXPECT_TRUE(IsPrime(2));
9     EXPECT_TRUE(IsPrime(3));
10    EXPECT_FALSE(IsPrime(4));
11    EXPECT_TRUE(IsPrime(5));
12 }
```

- Create some basic tests.

- The format is:

```
TEST(TestSuiteName, TestName) {
    ... test body ...
}
```

- You can find the basic macros at
(e.g., EXPECT_EQ, EXPECT_TRUE)
<https://google.github.io/googletest/reference/assertions.html>

GoogleTest

```
~/c/googletest cmake -S . -B build
-- The C compiler identification is GNU 10.3.0
-- The CXX compiler identification is GNU 10.3.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found Python: /usr/bin/python3.6 (found version "3.6.9")
   found components: Interpreter
-- Looking for pthread.h
-- Looking for pthread.h - found
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Failed
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Configuring done
-- Generating done
-- Build files have been written to: /home/jaechan/code/googletest/build
```

- Run the following command in the 'googletest' directory
- *cmake -S . -B build*
- If you don't have cmake, install it.

GoogleTest

- On success, you will have the directory 'build' within your googletest directory.

```
~ /c/googletest ls
basic_test.cc  build  CMakeLists.txt  memo.txt

~ /c/googletest ls build
'basic_test[1]_include.cmake'  CTestTestfile.cmake
bin                            _deps
CMakeCache.txt                 lib
CMakeFiles                     Makefile
cmake_install.cmake
```

GoogleTest

```
~ /c/googletest cmake --build build
[ 10%] Building CXX object _deps/googletest-build/googletest/CMakeFiles/gtest.dir/src/gtest-all.cc.o
[ 20%] Linking CXX static library ../../lib/libgtest.a
[ 20%] Built target gtest
[ 30%] Building CXX object _deps/googletest-build/googletest/CMakeFiles/gtest_main.dir/src/gtest_main.cc.o
[ 40%] Linking CXX static library ../../lib/libgtest_main.a
[ 40%] Built target gtest_main
[ 50%] Building CXX object CMakeFiles/basic_test.dir/basic_test.cc.o
[ 60%] Linking CXX executable basic_test
[ 60%] Built target basic_test
[ 70%] Building CXX object _deps/googletest-build/googlemock/CMakeFiles/gmock.dir/src/gmock-all.cc.o
[ 80%] Linking CXX static library ../../lib/libgmock.a
[ 80%] Built target gmock
[ 90%] Building CXX object _deps/googletest-build/googlemock/CMakeFiles/gmock_main.dir/src/gmock_main.cc.o
[100%] Linking CXX static library ../../lib/libgmock_main.a
[100%] Built target gmock_main

~ /c/googletest
```

- Build your test cases with the following command
- `cmake --build build`

GoogleTest

```
~ /c/googletest cd build
~ /c/g/build ctest
Test project /home/jaechan/code/googletest/build
  Start 1: FactorialTest.HandlesZeroInput
1/4 Test #1: FactorialTest.HandlesZeroInput ..... Passed
   0.00 sec
  Start 2: FactorialTest.HandlesPositiveInput
2/4 Test #2: FactorialTest.HandlesPositiveInput ... Passed
   0.01 sec
  Start 3: PrimeTest.HandlesOneInput
3/4 Test #3: PrimeTest.HandlesOneInput ..... Passed
   0.00 sec
  Start 4: PrimeTest.HandlesPositiveInput
4/4 Test #4: PrimeTest.HandlesPositiveInput ..... Passed
   0.00 sec

100% tests passed, 0 tests failed out of 4

Total Test time (real) =  0.02 sec
~ /c/g/build
```

- Go to the 'build' directory
- Execute the command ***ctest***
- You'll see that all your basic tests has passed.
- From here on, add more tests and try out something new by yourself.
- Refer to the document for more information.

GoogleTest

- ***The following should be tested through Google Test.***
 - File Initialization
 - When a file is newly created, a file of 10MiB is created and The number of pages corresponding to 10MiB should be created. Check the "Number of pages" entry in the header page.
 - Page Management
 - Allocate two pages by calling `file_alloc_page()` twice, and free one of them by calling `file_free_page()`. After that, iterate the free page list by yourself and check if only the freed page is inside the free list.
 - Page I/O
 - After allocating a new page, write any 4096 bytes of value (e.g., "aaaa...") to the page by calling the `file_write_page()` function. After reading the page again by calling the `file_read_page()` function, check whether the read page and the written content are the same.

Milestone & DEADLINE

- Milestone 1
 - Implement disk space manager and submit a report (Wiki) including your design.
 - Your report should include
 1. Overall descriptions about the disk space manager APIs.
 2. The description of unittests.
 - Deadline: Oct 01 11:59pm
- Milestone 2-3 will be announced sometime next week.
- We'll only score your commit before the deadline and your submission after that deadlines will not be accepted.

Thank you
