Code Quality Report Team 11

Readability

We used Artistic style program to automatically format all our code to the same coding standard. We used the GNU coding standard since every group member felt comfortable with this standard since we learned and used it during the course.

All of our functions are no longer than 20 lines. The code never ran more than 2 consecutive statements at once. We reused code that appeared more than once in the program. One example of this is the insert() function which inserts an object into our global linked list. All static functions that refmem.c uses are declared in the top of the file. The refmem.h functions are written using encapsulation which makes each function easy to understand.

```
obj *allocate(size_t bytes, function1_t destructor)
{
    obj *data = calloc(1, (sizeof(objectInfo_t) + bytes) );

    objectInfo_t *objectToReturn = data;
    data = data + sizeof(objectInfo_t);


    objectToReturn->rf = 0;
    objectToReturn->func = destructor;
    objectToReturn->size = bytes;


    //c3 == 0, c3->cell = 1
    insert(objectToReturn);

    return data;
}

obj *allocate_array(size_t elements, size_t elem_size, function1_t destructor)
{
    obj *data = calloc(elements, ((sizeof(objectInfo_t)/elements) +1 + elem_size))

    objectInfo_t *objectToReturn = data;
    data = data + sizeof(objectInfo_t);

    objectToReturn->func = destructor;
    objectToReturn->size = elem_size*elements;
    objectToReturn->rf = 0;



    insert(objectToReturn);

    return data;
}

[]
void retain(obj *c)
{
    if(c == NULL) {}
    else
    {
        objectInfo_t *objectInfo = c - sizeof(objectInfo_t);
        objectInfo->rf = objectInfo->rf+1;
    }

}
```

Figure 1: Example of code from refmem.c

Each function is well capsulated and short, the data structures and program design is explained in a structured way. The variable names explains the variables in a coherent and clear way which makes the code more readable. This makes it good for programmers who did not design the program to maintain the library.

Profiling

We integrated our code into assignment two from IOOPM19. We used gprof on the original test file from assignment two and gprof on the same file but after we implemented our memory management system. The results from gprof shows how long time the testfile ran and how each function performed relative to eachother:

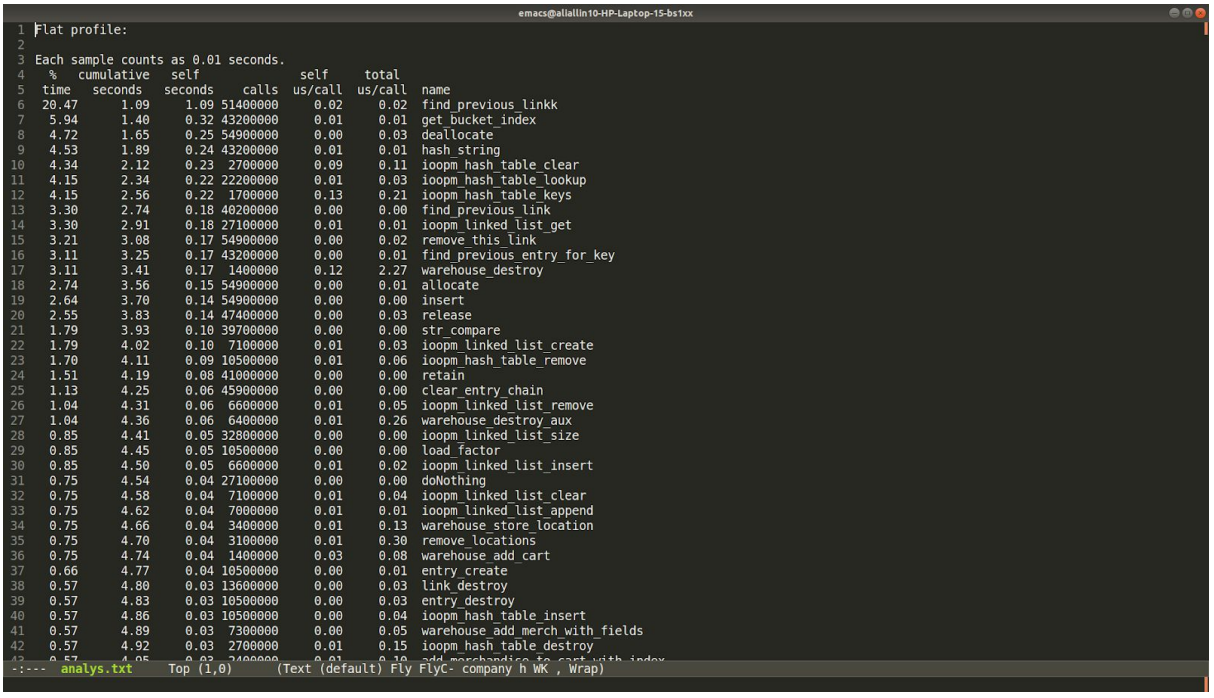

Figure 2: gcov profiling from assignment 2 tests using our library

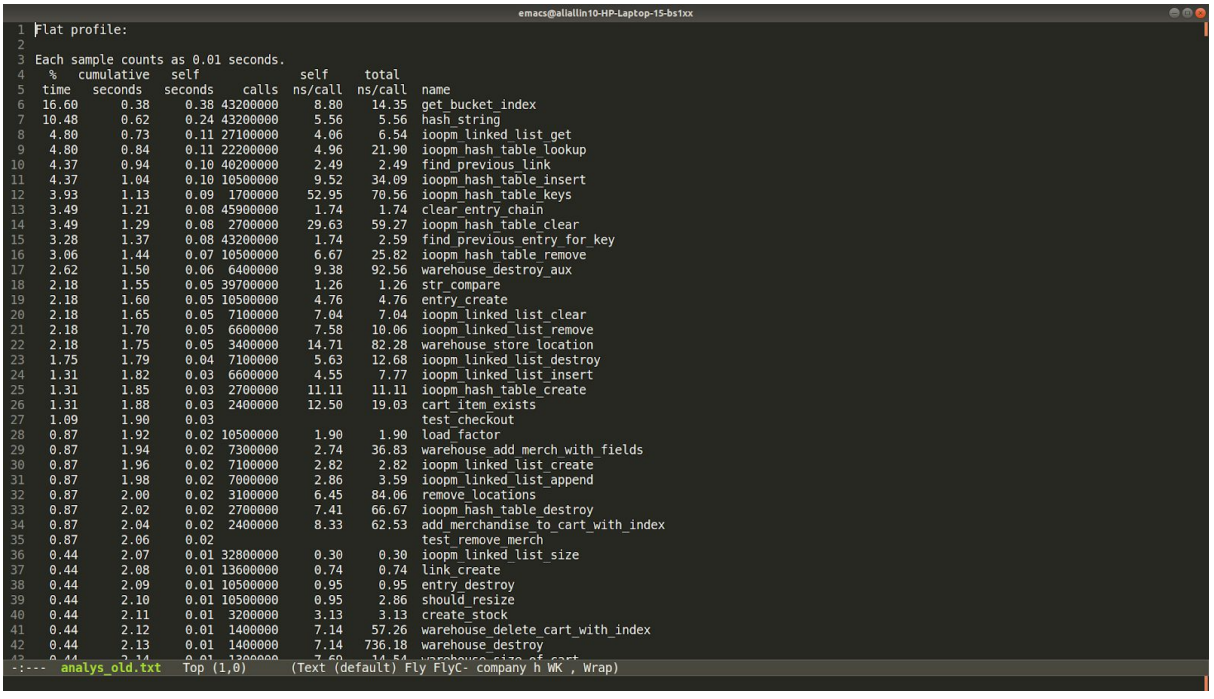

Figure 3: gcov profiling from assignment 2 tests without our library

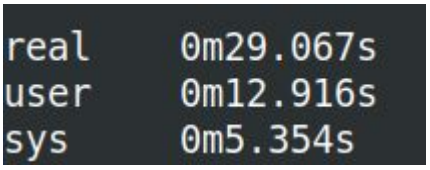

Figure 4 : Total time spent from the assignment 2 tests without our library

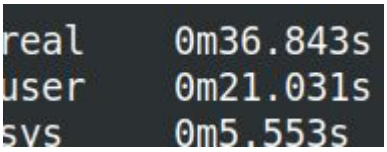

Figure 5: Total time spent from the assignment 2 tests using our library

From the pictures above we can see that integrating our memory management system into assignment 2 made the program run 26% slower. From figure 1 we can see that the find_previous_link from our library had the biggest impact on the performance difference, ending up running 20% of the time. This function is used often because its used every time we want to remove an object. And since we remove 1 object for every time we run an object it ran a lot. It's also a slow function because it searches an entire linked list for every time it's called making the time complexity (n^2).

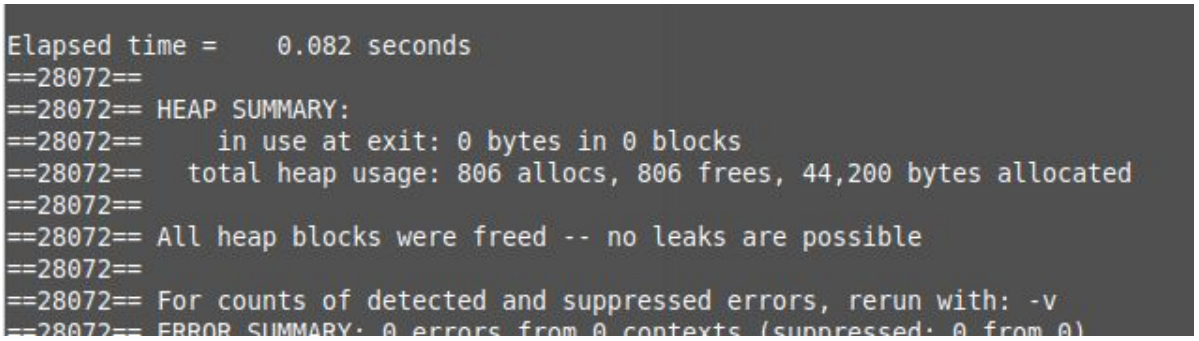

Figure 6: Amount of blocks allocated from assignment 2 tests using our library

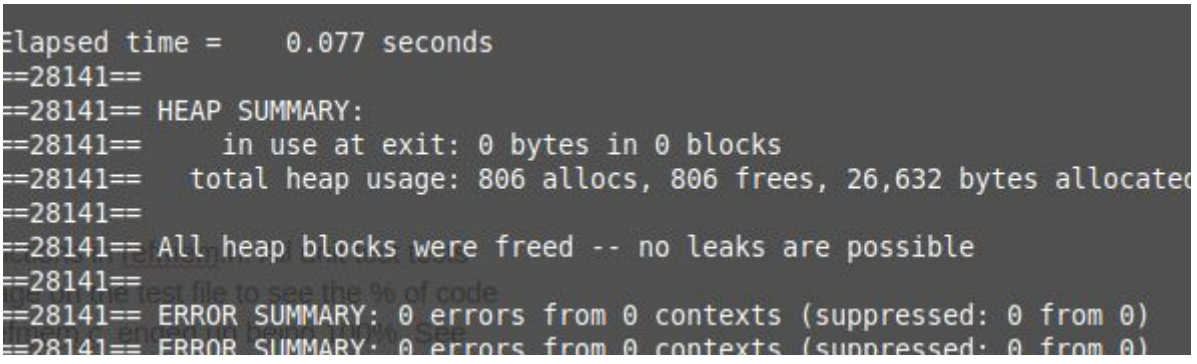

Figure 7: Amount of blocks allocated from assignment 2 tests without using our library

From figure 6 and 7 we can see that our implementation allocated 66 % more memory. The specification of this program said we should be <100% which means the program met up with the requirements for this example since the objects allocated are quite large. However if we were to allocate a struct with the size of 8 bytes our implementation would still allocate 32

bytes which is 400% of the original memory usage. Unfortunately we couldn't figure out a solution for this

Code Coverage

We managed to use CUnit to create unit tests for all functions in refmem.h. All unit test tests passed and had 0 memory leaks. We used code coverage on the test file to see the % of code ran from the refmem.c library. The code coverage for refmem.c ended up being 100%. See picture below:

```
Elapsed time =    0.000 seconds
gcc -g -Wall -std=c11 -fprofile-arcs -ftest-coverage --coverag
./memory


     CUnit - A unit testing framework for C - Version 2.1-3
     http://cunit.sourceforge.net/


Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      4      4    n/a      0        0
               tests     14     14     14      0        0
             asserts     63     63     63      0      n/a

Elapsed time =    0.000 seconds
gcov -b test.c
File 'proj/test.c'
Lines executed:100.00% of 317
No branches
Calls executed:100.00% of 212
Creating 'test.c.gcov'

File 'proj/refmem.c'
Lines executed:100.00% of 124
Branches executed:100.00% of 44
Taken at least once:86.36% of 44
Calls executed:100.00% of 13
Creating 'refmem.c.gcov'
```

Figure 8: Code coverage for refmem.c used ran with tests.c