Preparation

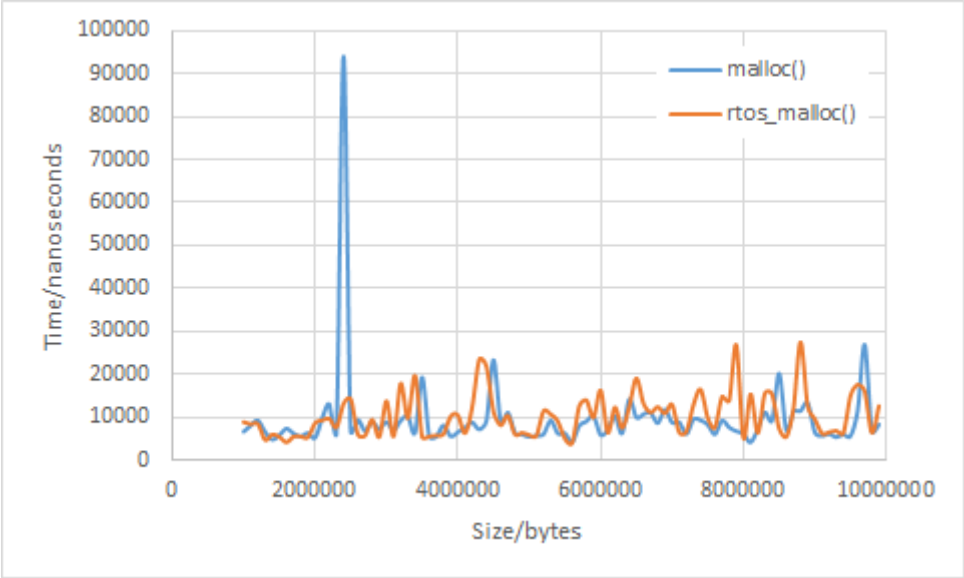
Following the instruction, I wrote a timer function to count the time conveniently for different functions:

```
#include<stdio.h> /*for printf */
#include<stdint.h> /*for uint64*/
#include<time.h> /*for clock_gettime */
#include <fstream> // store the data into a file

#include "rtos-alloc.h"


/**
 * timer for rtos_malloc() and malloc()
 * @param   fct(size_t)   rtos_malloc() or malloc()
 * @param   size    size
 * @return  running time [nanosecond]
 */
size_t Timer(void*  fct(size_t),size_t size)
{
    size_t execTime;/*timeinnanoseconds*/
    struct timespec tick, tock;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&tick);
    /*do stuff*/
    fct(size);

    clock_gettime(CLOCK_PROCESS_CPUTIME_ID,&tock);
    execTime=1000000000*(tock.tv_sec - tick.tv_sec)+tock.tv_nsec - tick.tv_nsec;
    printf("%llu \n",(long long unsigned int)execTime);
    return execTime;
}

int main()
{
    //Open file in write mode
    std::ofstream outfile;
    outfile.open("time.txt");
    //Write the header
    outfile << "Size malloc() rtos_malloc()\n";

    for(size_t size = 10e5; size < 10e6; size+=10e4)
    {
        outfile << size << ' ' << Timer(malloc, size) << ' ' <<
Timer(rtos_malloc, size) << '\n';
    }

    outfile.close();

    return 0;
}
```

I export the data into a .txt file, which is easy to import into Excel and plot.

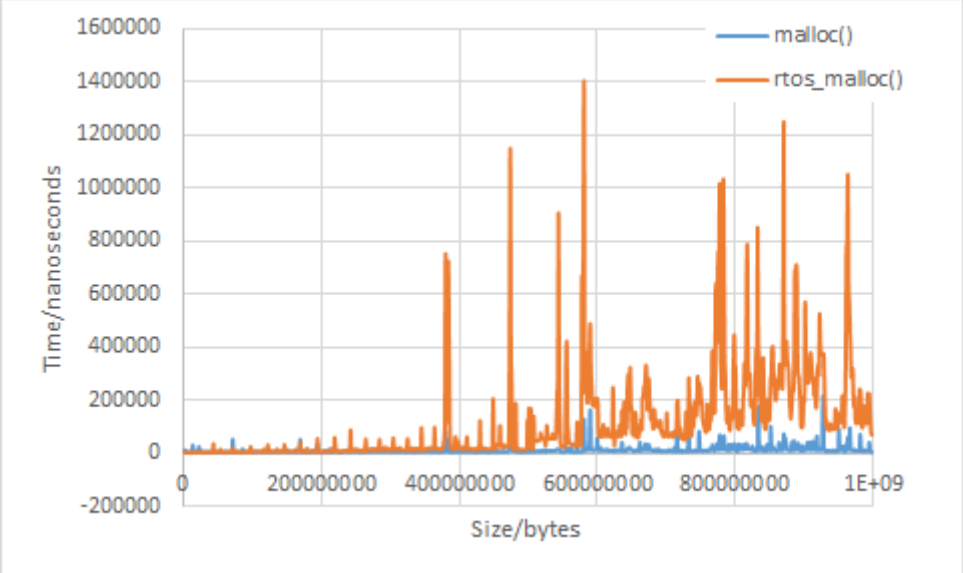
Analysis

First, I tried a group of relatively small sizes, from 10e5 to 10e6 with step 10e4. Here is the result:



Although there is a spike for malloc(), it doesn't influence our judgement. The performance of both functions are similar. However, there is no specific relationship between the time and the the size.

Then I tried larger size, from 10e5 to 10e8 with step 10e5.



Now, things have changed. `rtos_malloc()` shows heavy disturbance, while `malloc()` is more stable. It might be caused by the following reasons:

1. Process-switching. OS may de-schedule the `rtos_malloc()` process and schedule other processes.
2. Concurrency. `rtos_malloc()` doesn't provide concurrency protection mechanism.