Exercise 1: Time and Memory

In this exercise you will be introduced to some concepts that are important for C-programming and necessary for this course, even if it is not directly related to real-time.

In the exercises in this course, only parts of the information you need is given in the exercise text. You will also need to find information on the internet and other sources. Remember to also ask the student assistants, they are there to help you.

1. Time

In this task, you will compare the behavior of different ways of measuring time.

1.1. Measuring time

There are different methods that you can use to measure time, depending on what you want to measure and how accurate you need the measurements to be.

The first kind of time measurements are ones that return *wall time*, which sounds like a wonderfully practical thing that returns "real time", but since the clock on the computer could be wrong then automatically fixed by the OS, or daylight savings time changes, or even leap seconds, the result is that these kinds of functions can report unexpected discontinuities in time.

For real-time applications, we are generally only interested in periods and durations, so we instead prefer using *monotonic* clocks, which are clocks that will always count upward at a constant rate. The absolute value of these clocks doesn't mean much, and is usually just the number of ticks since the system boot time.

If you want to measure the time it takes to execute a program, you can use a small program called `time`, as follows:

```
time ./your_program
```

If you want to measure time within your program, you can use these methods:

- `clock_gettime()`, which has capabilities for both `REALTIME` (which is actually wall time) and `MONOTONIC`, as well as some others.
- `times()`, which reports the user and system time spent executing this program in hundredths of seconds. Alternatively `getrusage()`, which also reports the resource usage for many other things.
- The CPU instruction `rdtsc` (which is available as `__rdtsc()` via `x86intrin.h`), which reads the CPU Time Stamp Counter directly. On the computers at the lab, the TSC ticks at the same frequency as the rest of the CPU, at 2.66 GHz.

Here are some useful functions for manipulating `struct timespec`, used by `clock_gettime()`:

```
struct timespec timespec_normalized(time_t sec, long nsec){
    while(nsec >= 1000000000){
        nsec -= 1000000000;
        ++sec;
    }
    while(nsec < 0){
        nsec += 1000000000;
        --sec;
    }
    return (struct timespec){sec, nsec};
}

struct timespec timespec_sub(struct timespec lhs, struct timespec rhs){
    return timespec_normalized(lhs.tv_sec - rhs.tv_sec, lhs.tv_nsec - rhs.tv_nsec);
}

struct timespec timespec_add(struct timespec lhs, struct timespec rhs){
    return timespec_normalized(lhs.tv_sec + rhs.tv_sec, lhs.tv_nsec + rhs.tv_nsec);
}

int timespec_cmp(struct timespec lhs, struct timespec rhs){
    if (lhs.tv_sec < rhs.tv_sec)
        return -1;
    if (lhs.tv_sec > rhs.tv_sec)
        return 1;
    return lhs.tv_nsec - rhs.tv_nsec;
}
```

1.2. Functions for waiting

If we want our code to wait for a given amount of time we use a sleep function, which can be `sleep()`, `usleep()` or `nanosleep()` depending on the duration of the wait. While one thread is sleeping, others can run.

Another method for delaying the execution is to use busy-wait delays, which means that the thread will execute a function that takes a known amount of time. Below is a sample code for a busy-wait delay function:

```
void busy_wait(struct timespec t){
    struct timespec now;
    clock_gettime(CLOCK_MONOTONIC, &now);
    struct timespec then = timespec_add(now, t);

    while(timespec_cmp(now, then) < 0){
        for(int i = 0; i < 10000; i++){}
        clock_gettime(CLOCK_MONOTONIC, &now);
    }
}
```

Several of these functions are not technically part of the C standard library, but rather the GNU extended standard library. Remember to set the compiler flag `-std=gnu11`.

Task A:

Write a simple program that just waits for one second before terminating. Use the command-line program `time` to compare the real, user, and system time passed when waiting by using

- `sleep()` (or `usleep()` / `nanosleep()`)
- `busy_wait()` with `clock_gettime(CLOCK_MONOTONIC,...)` (provided above)
- `busy_wait()` with `times()` (which you will have to make yourself)

Which one uses the most system time, and why?

Task B:

Estimate and compare the access latency (the time it takes to get a value from a timer) and the resolution (the shortest measurable time interval) for these three timers:

- `__rdtsc()`
- `clock_gettime()`
- `times()`

You can estimate the access latency by reading the timer `n` times in a `for` loop, then dividing the time taken (measured with `time`) by `n`:

```
for(int i = 0; i < 10*1000*1000; i++){
    // read timer
}
```

You can estimate the resolution of a timer by finding the typical (or shortest) time difference between two successive calls to the timer. Remember to convert the time difference into nanoseconds. Create a histogram with `gnuplot` by using the sample code below:

```
int ns_max = 50;
int histogram[ns_max];
memset(histogram, 0, sizeof(int)*ns_max);

for(int i = 0; i < 10*1000*1000; i++){

    // t1 = timer()
    // t2 = timer()

    int ns = // (t2 - t1) * ??

    if(ns >= 0 && ns < ns_max){
        histogram[ns]++;
    }
}

for(int i = 0; i < ns_max; i++){
    printf("%d\n", histogram[i]);
}
```

Pipe the output from the program into `gnuplot` like this:

```
./program_name | gnuplot -p -e "plot '<cat' with boxes"
```

Task C:

Measure the time it takes to context-switch to the kernel and back, by inserting a call to `sched_yield()` (found in `sched.h`) between the two successive calls to the timer. You only have to do this for a single timer (`clock_gettime()`), and you will have to increase `ns_max`.

2. Memory

Dynamic memory allows memory to be allocated while the program is running. In C dynamic memory allocation uses the functions malloc() and free(). This is typically used when the required memory size for a program is not known when it is compiled.

It is important to free all the memory you have allocated, or you can get a memory leak. This means that as the program runs it takes more memory than it returns, which over time will crash the system when it is no more memory available. Dynamic memory is often avoided for real-time systems because of the danger of memory leaks and that it increases the complexity of the code. But it is still often needed, especially in communication handling.

Task D:

The program below allocates memory for a very large 8GB matrix of 64-bit integers:

```
long xy_size    = 1000*1000*1000;       // 8 GB (sizeof(long) = 8 bytes)
long x_dim      = 100;
long y_dim      = xy_size/x_dim;

long** matrix   = malloc(y_dim*sizeof(long*));

for(long y = 0; y < y_dim; y++){
    matrix[y] = malloc(x_dim*sizeof(long));
}

printf("Allocation complete (press any key to continue...)\n");
getchar();
```

The computers at the lab have 8GB of RAM, which means there shouldn't be enough space to allocate this matrix. Open the System Monitor from Dash (aka the "start menu"), and go to the resources tab. From here you can see the total memory used by the system.

Run the program, and observe the effects on the memory usage. Don't worry if the computer becomes unresponsive, it's just old and needs some time. What happens? What is the difference between "Memory" and "Swap"?

Now change the shape of the matrix by increasing `x_dim` first to 1000, then 10000, and run the program again. What happens?

Run the program with `x_dim = 10000` again, but this time write zeros to all the memory:

```
memset(matrix[y], 0, x_dim*sizeof(long));
```

Explain why something different happens when the memory is also cleared.

Task E:

A common application of dynamic memory is to have a list that will grow and shrink in size as the program runs. The beginnings of the code for a simple dynamic array is available from the same folder where you downloaded this exercise text on Blackboard.

Implement `array_insertBack()`, but without implementing a way to grow the size of the array if there isn't enough capacity (we'll get back to that later).

Create an array with some small capacity, then `insertBack` more elements than there is capacity for, and run the program. What happens?

Compile the program again, but now add the compiler flags `-g -fsanitize=address`.

AddressSanitizer is an invaluable tool for making sure you don't get your pointers and memory all tangled up. If you are writing code that uses pointers in any non-trivial way, you should probably enable this memory error detector – as long as the platform supports it. And if it doesn't, see if you can copy the relevant code to a platform that does *while you work on it, then port it back after it has been tested.*

Task F:

Implement `array_reserve()`, such that it can grow the capacity of the array. You will have to relocate the existing data by first allocating new space, moving the existing data, and then freeing the old data.

Fix `array_insertBack()` so that it detects a lack of capacity and reserves more if necessary. Run and verify that the code works.

Task G:

Typically when increasing the capacity of a dynamic array, you would increase the capacity by some factor, say 2x or 1.5x.

Let's say our array currently has a capacity of 2 elements. From here, we continuously insert new elements at the back, increasing the capacity as necessary. Draw out (or otherwise visualize) what would happen to the available memory if each new relocation requires a contiguous section of memory that is 2x the size of the previous section. Then do the same thing for 1.5x.

Devise and implement a test to verify your hypothesis. Hint: You can print pointers with the format specifier `%p`. Hint 2 (spoilers): It probably won't work as you expect.