# TTK4147 - Real-time Systems
# Exercise 1 - Time

September 2020

Remember to check in at the lab, https://innsida.ntnu.no/checkin/room/3066, to allow for easier infection control in the event of a Covid-19 infection on campus.

In the first exercises 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. This exercise focuses on the behavior of different methods for waiting and measuring time. Only parts of the information you need in the exercises is given in the exercise text. You will also need to find information on the internet and from other sources, or ask the student assistants, they are there to help you.

The computers on the real-time lab have two different operating systems; Windows and Ubuntu. When the computer boots, a menu where you can select between these is presented. Select `Ubuntu` for this exercise, as you will use the GNU/Linux environment.

If you have questions or want to get your assignment approved, please apply for a spot (using your NTNU-user) in the queue here: https://s.ntnu.no/rts.

## 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 the clock on the computer could be wrong due to adjustments made by the OS, changes in daylight savings time 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 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 one of 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 Time Stamp Counter (TSC) ticks at the same frequency as the rest of the CPU, 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;
}
```

## 2 Waiting

If we want our code to wait for a given amount of time we use a sleep function, some examples are: `sleep()`, `usleep()` or `nanosleep()`. 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()`, `usleep()` or `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"
```

If gnuplot is not installed or you get an error regarding gnuplot, use the following command to install the correct version:

```
$ sudo apt-get install gnuplot-x11
```

# 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`.