# Exercise 1: Memory, Errors and Time

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. This is because part of what you need to learn is how to find information on the Internet and other sources. Boot up in Linux.

## 1. Pointers and dynamic memory allocation

Pointers and dynamic memory allocation are important concepts in the C programming language. If pointers are a new concept, you should probably read more about it in a book or search for it on the Internet, as it will not be explained in detail here.

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.

**Assignment A:**

The function given below allocate a memory area, large enough to store 100000 integers. After the allocation, it will try to write and read an integer to the area.

```
void allocate(int value) {
      int *ptr = NULL;
      ptr = malloc(100000 * sizeof(int));
      *ptr = value;
      printf("test of allocated memory: %i\n");
}
```

Create a program that will run this function indefinetely. What happens and why? Run the program `top` in another shell. What happens to the memory usage of your program.

### 1.1. Linked list

A common application of dynamic memory is to have a list that will grow and shrink in size as the program runs. This could be a buffer queue. For this we use a linked list, where each element in the list knows the element before and after itself.

**Assignment B:**

Download *linked_list.h, linked_list.c* and *list_test.c* from it's learning. The linked list header file contains definition and comments describing these, while the list test file contain a little program for testing the functionality. Your task is to make a Makefile for the project or import it to Eclipse, and then complete the program by implementing the functions in the linked list C file. You should use the program *valgrind* to check that there are no memory problems.

```
valgrind ./list_test
```

When running the list_test program, it should print something like this (depending on how your `list_print()` function works).

```
create list
append valued 0 to 9
0 1 2 3 4 5 6 7 8 9
list sum: 45
insert value in the middle of the list
0 1 2 3 4 99 5 6 7 8 9
get inserted value: 99
0 1 2 3 4 99 5 6 7 8 9
extract inserted value: 99
0 1 2 3 4 5 6 7 8 9
remove all but the last value
9
remove the last value

delete list
```

## 2. Detecting Errors from Return Values

All functions in C (except void functions) will return a value or a pointer. A function called `add(int a, int b)` would typically return the sum of `a` and `b`. A function that returns true or false will typically return 1 if true or 0 if false.

Return values can also contain information whether the function executed successfully or not. A failure will typically return a negative value or a NULL pointer. It is important to check the return values of all functions that can fail. An undetected failure can give strange and difficult to debug problems with the program. Therefore, you should always check the return values when programming in C. Be aware that a 0 can both mean "success" and "false", depending on the function. This can be confusing, so you should always check man pages to be sure.

In addition to the return values, there is a variable called `errno` that stores the last error message. You will normally not interact with this variable directly, but it is used by the `perror()` function.

**Assignment C:**

The function given in assignment A did not check the return value from `malloc()`. You should change the `allocate()` function so it will detect an error from `malloc()`. When an error is detected, the program should print a text describing the error with `perror()` and exit the program.

# 3. Time

Time is obviously important in real-time systems, and it is useful to be able to know the current time, measure time etc. Some basic concepts of time in computer programs will be covered here.

## 3.1. 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. 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 either use `gettimeofday()` or `times()`, depending on that you want to measure.

- `gettimeofday()` gives you the current system time. If you read the system time at two points in time, you can compare them and measure how much time has passed in the "real world". This is also called absolute time.
- `times()` gives you the number of CPU cycles that have been used by the thread or process. How many CPU cycles that have been used is a measurement of how much time the CPU have spent executing the thread of process. This is called execution time.

Use `man` for details on these functions and how to use them. There are no exercises for these functions directly, but they can be useful in future exercises.

## 3.2. Functions for waiting

### Sleep

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.

### Busy-wait delay

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.

```
#include <sys/times.h>
void busy_wait_delay(int seconds)
{
      int i, dummy;
      int tps = sysconf(_SC_CLK_TCK);
      clock_t start;
```

```
	struct tms exec_time;

	times(&exec_time);
	start = exec_time.tms_utime;

	while( (exec_time.tms_utime - start) < (seconds * tps))
	{
		for(i=0; i<1000; i++)
		{
			dummy = i;
		}
		times(&exec_time);
	}
}
```

**Assignment D:**

Create two threads, that each prints a message, sleeps for 5 seconds and then prints a new message. Run the program using the *time* application.

Modify the same program to use the busy wait delay instead of sleep, and run it. How long does it take to execute the program, why is there a difference? How does this demonstrate absolute time and execution time? Run this multiple times and observe the results.

> *You must add `-lrt` to the linker flags in your `makefile` to link your program with the library containing some of the time functions.*

## 4. Approving the exercise

1. Run list test with your implementation of the linked list.
2. Run the program from assignment C that checks for errors.
    - Explain what happened when you tried to write to the variable when not checking the return value of `malloc()`.
3. Run the program from assignment D, using both sleep and busy-wait.
    - Explain the difference between absolute time and execution time?
    - Why does these two programs use different amount of time (absolute time)?