In this exercise we familiarize ourselves to asynchronous i / o and signals.

Exercise 8a (Multiplexed i/o and asynchronous i/o, 1p)

In this exercise you use a so called multi sensor simulator. Multi sensor simulator simulates a measuring system having many sensors (in this case 10 sensors). Each sensor is measuring a temperature in a certain point in some industrial process. Temperatures can vary between 0 and 500. The multi sensor simulator creates the measurement values. Each sensor sends it values to their own descriptor so that you have 10 sensor descriptors in your program.

Each sensor makes it’s own decision, when they send a new value to the descriptor. The delay between two values from the same sensor can vary from very short time (about a few milliseconds) to 10 seconds. Your program needs to receive each value as fast as possible after it has been sent from the sensor. All sensors send the same number of values. So it can be big differences between sensors how long it takes for them to send all values. Number of values per sensor is indicated with the parameter when StartSimulator function is called.

When sensor has sent all values, reading the file descriptor of that sensor causes end of file situation. The termination of the program must be based on this fact and not on the number of values received.

To help to analyze the response times the sensors send a sending time with the value. This means that you can verify how much time has elapsed between sending and receiving the value. Sensors send the data to the file descriptor as a structure that is defined in the following way:

```
      typedef struct {
            struct timespec moment;
            int value;
     } Tmeas;
```

Immediately after you have read this kind of structure from the sensor file descriptor you can read the current time from the computer and calculate the time difference that indicates the delay of your read. Use the function `diff_timespec` for that purpose (see MultiSensorSimulator.h). Calculate and display the sum of all delays because it gives a good overall figure how real time your application works. Sum of delays can be calculated with the function `increment_timespec` (see MultiSensorSimulator.h)[1].

It is easy to use the multi sensor simulator. Only thing you need to do is to include the file MultiSensorSimulator.h to your source code and link object file MultiSensorSimulatorEdu.o to your object code, if you are working with Edunix computer. The files are given at Tuubi.

[1] Struct timespec has two fields, tv_sec and tv_nsec that must be reset to zero before `increment_timespace` is used to increment the time value.

If the source file of your application were myapplication.c, then you can compile and link it in the Edunix in the following way:

```
gcc -o myapplication.exe myapplication.c MultiSensorSimulatorEdu.o -lrt
```

In the source code you only need to call function StartSimulator as follows:

```
int sensorDescriptors[10];
StartSimulator(sensorDescriptors, 5); // Use 5 or 10
// Generates 5 values per descriptor
```

After doing that you can read measurement values from sensors. For example, you could read one measurement value from the first sensor and display it to the screen in the following way:

```
Tmeas measurement;
read(sensorDescriptors[0], &measurement, sizeof(Tmeas));
printf("Measurement value was %d\n", measurement.value);
```

Do one of the following options.

Option A. (Multiplexed i/o)

Use multiplexed i/o solve the problem above.

Option B. (Asynchronous i/o, "new" Posix way)

Solve the same problem using asynchronous i/o.

Remark 1. Use new style of asynchronous i/o. This means that you need to use aio_xxxx functions and need a signal handler.

Exercise 8b (Extra exercise, 0.25p)

You can do another option as an extra exercise and compare the response time delays between the methods.

Exercise 8c (Extra exercise, 0.25p)

There were two problems in exercise 7B.

The first problem was that the program hung (got stuck) sometimes. You fixed that problem by calling one of the wait functions in a loop in the signal handler. The second problem left unresolved.
The second problem was that some outputs were missing. The reason for this was the race condition. Sometimes the next signal was delivered before the program had reached the pause again.

Use `sigsuspend` instead of `pause` to prevent the race condition. Now you have done two fixes. But still the program at least sometimes misses the output. What is the reason?

What you need to do to make it work perfectly? Do the final modification and test it so many times that you can be sure that it works in a reliable way. Is `sigsuspend` necessary or useful in the final solution?