Waiting time The two functions presented in the Table 1 are more or less equivalents, they both wait for a number $t$ of microsecond before returning. What are the tradeoffs between the two designs?

```c
void thread_pause1(int t) {
    struct timeval start, current;
    long diff;
    gettimeofday(&start, NULL);
    do {
        gettimeofday(&current, NULL);
        diff = (current.tv_sec - start.tv_sec) * 10^6 + (current.tv_usec - start.tv_usec);
    } while(diff < t);
}
```

```c
void thread_pause2(int t) {
    struct timeval start, current;
    long diff;
    gettimeofday(&start, NULL);
    do {
        if( 0 != usleep(t) ) {
            gettimeofday(&current, NULL);
            diff = (current.tv_sec - start.tv_sec) * 10^6 + (current.tv_usec - start.tv_usec);
            t = t - diff;
        } else {
            t = 0;
        }
    } while(t > 0);
}
```

Table 1: Two implementations of the microsecond pause.

Concurrent accesses The following code is executed in parallel by two threads in the context of the same application. What are the possible final values for the variable $\text{global}_v$ if the original value was 10? Can you generalize to $n$ threads executing concurrently the code?

```c
int x = global_v;
x = x + 10;
\text{global}_v = x;
```

First In First Out FIFO is the acronym for First In, First Out. It describe the principle of a queuing system, where the first what come first is served first, and what come after have to wait until everything that arrive before is served. It is more than similar with the waiting line we encounter every day in our life.

In the figure 1 (a) we have a design of a unbounded FIFO. On the (b) part the push operations is showed. The new element (in gray) is pushed to the bottom of the FIFO (and it will be returned only after all previously posted elements have been returned). On the (c) part the pop operation is described. The oldest element in the FIFO is returned while the gray element added in the (b) part is still the newest in the FIFO.

Thread-safe FIFO The first part of the homework consist of implementing a thread-safe FIFO. You will have to create a library, providing 4 functions. The architecture we’re targeting is the x86 32 bits. There are no restrictions on what language (C/C++ or assembly) or synchronization primitives (any synchronization
Figure 1: A sketch for a FIFO with push and pop operations.

primitives provided by the POSIX norm or even atomic operations) you’re using. The delivered library will be tested first for correctness, and then for performance on a heavily threaded application.

typedef struct __fifo_elem_t {
    struct __fifo_elem_t* prev;
    struct __fifo_elem_t* next;
} fifo_elem_t;

Figure 2: Description of the elements in the FIFO

The library should contain four functions. One for initializing the FIFO, one for queuing a new element, one for dequeuing an element and one for destroying the FIFO. The FIFO will contain elements based on the structure described in the 2.

1. **fifo_create** will allocate memory for a FIFO and return it to the application. It should initialize the FIFO to a state where any operations on the FIFO will succeed. If anything wrong happens during the initialization NULL should be returned.

2. **fifo_push** will add one more element in the FIFO. There is no memory to be allocated and the element is coming directly from the application. This function should modify only the FIFO and the fifo_elem_t structure in order to make the element part of the FIFO. We suppose that each element is allowed to belong to only one FIFO at the time. The return value will reflect the fact

\[
\text{fifo_t* fifo_create( void );}
\]

\[
\text{int fifo_push( fifo_t* fifo, fifo_elem_t* elem );}
\]

\[
\text{fifo_elem_t* fifo_pop( fifo_t* fifo );}
\]

\[
\text{int fifo_destroy( fifo_t** fifo );}
\]

Figure 3: The 4 interface functions for the library (API).
that the element was inserted in the FIFO or not. A successful operation will return 0, while a failures should return any negative value and the element should not be considered as belonging to the FIFO.

3. `fifo_pop` will remove the *oldest element* from the FIFO and return it to the application. If the FIFO is empty (no more elements inside) this function should return NULL.

4. `fifo_destroy` will release the memory allocated for the FIFO and mark all elements still in the FIFO as being released. These elements will not be freed. The return value should be 0 if everything went smoothly and no elements were inside the FIFO. Any other positive value indicate the number of elements in the FIFO when the FIFO was destroyed. For everything else (fatal failures) the return value should be negative.