

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.

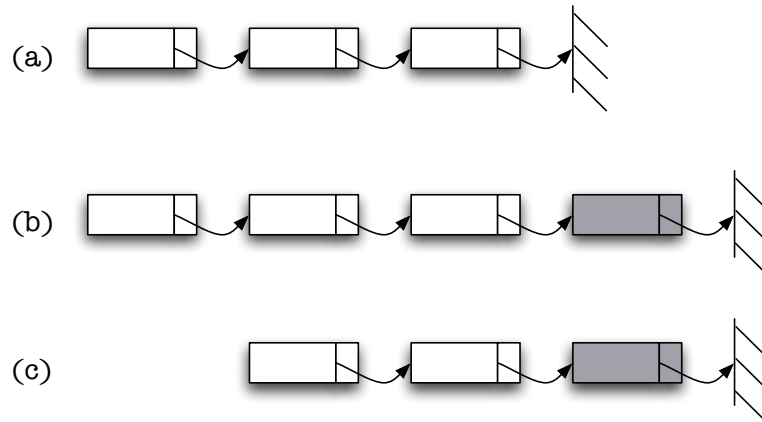


Figure 1: A sketch for a FIFO with push and pop operations.

In the figure 1 (a) we have a design of a unlimited FIFO. On the (b) part the push operations is showed. The new element (in gray) is pushed to the bottom of the FIFO as it has to wait before being able to be delivered that all prior elements are removed. 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.

1 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 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.

```
fifo_t*      fifo_create( void );
int          fifo_push( fifo_t* fifo, fifo_elem_t* elem );
fifo_elem_t* fifo_pop( fifo_t* fifo );
int          fifo_destroy( fifo_t** fifo );
```

Figure 3: The 4 interface functions for the library (API).

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 that the element was inserted in the FIFO or not. A successful operation will return 0, while any failures should return anything else 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.

2 MPI FIFO

For the second part of the homework a FIFO should be implemented as a MPI library. This version will only be tested for correctness. There are few differences:

1. The initialization function take one more argument, the communicator that will be used for the FIFO. Beware, this is one of the few functions that have a global scope (i.e. all nodes in this particular communicator have to call this function simultaneously).
2. The destruction function is the second global operation. All nodes have to be involved in the destruction.

3. The push and pop operations are not globals, but their result have to be visible for everybody in the communicator specified in the initialization step.

The **pop** and **push** operations can be implemented in several ways. The only requirement here is that they don't have to be globals, each node can **push** or **pop** at any moment. Choose the approach that seems best/simplest for you.

```
fifo_t*      mpi_fifo_create( MPI_Communicator comm );
int          mpi_fifo_push( fifo_t* fifo, fifo_elem_t* elem );
fifo_elem_t* mpi_fifo_pop( fifo_t* fifo );
int          mpi_fifo_destroy( fifo_t** fifo );
```

Figure 4: The 4 interface functions for the library (API).