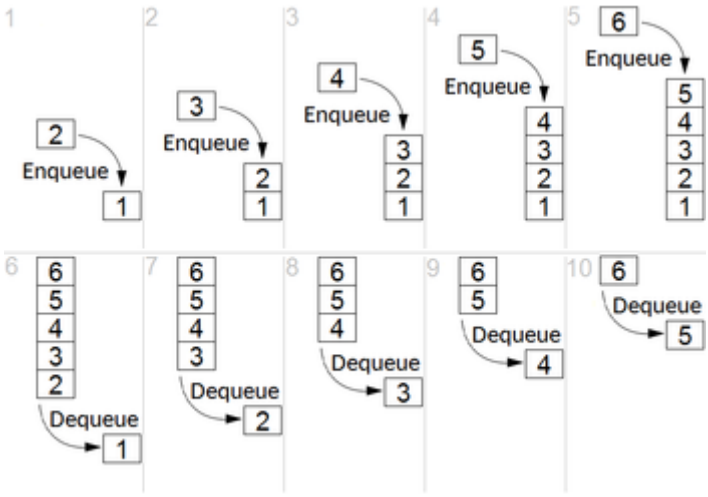
FIFO (computing and electronics)

FIFO – an acronym for **first in, first out** – in computing and in systems theory, is a method for organising the manipulation of a data structure – often, specifically a data buffer – where the oldest (first) entry, or 'head' of the queue, is processed first.



Representation of a FIFO (queue) with *enqueue* and *dequeue* operations.

Such processing is analogous to servicing people in a queue area on a first-come, first-served basis, in the same sequence in which they had arrived at the queue's tail.

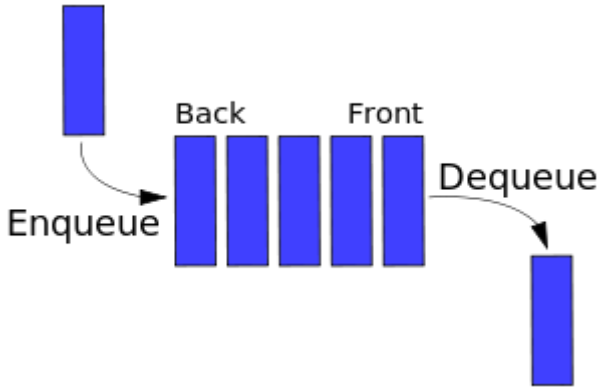
FCFS is also the jargon term for the FIFO operating system scheduling algorithm, which gives every process central processing unit (CPU) time in the order in which it is demanded. FIFO's opposite is LIFO, last-in-first-out, where the youngest entry or 'top of the stack' is processed first.[1] A priority queue is neither FIFO or LIFO but may adopt similar behaviour temporarily or by default. Queueing theory encompasses these methods for processing data structures, as well as interactions between strict-FIFO queues.

Contents

Computer science

Data structure

Depending on the application, a FIFO could be implemented as a hardware shift register, or using different memory structures, typically a circular buffer or a kind of list. For information on the abstract data structure, see Queue (data structure). Most software implementations of a FIFO queue are not thread safe and require a locking mechanism to verify the data structure chain is being manipulated by only one thread at a time.



Representation of a FIFO (first in, first out) queue

Code

The following code shows a linked list FIFO C++ language implementation. In practice, a number of list implementations exist, including popular Unix systems C sys/queue.h macros or the C++ standard library std::list template, avoiding the need for implementing the data structure from scratch.

```
#include <memory>
#include <stdexcept>

using namespace std;

template <typename T>
class FIFO {
    struct Node {
        T value;
        unique_ptr<Node> next = nullptr;

        Node(T _value): value(_value) {}
    };

    unique_ptr<Node> front = nullptr;
    unique_ptr<Node>* back = &front;

public:
    void enqueue(T _value) {
        *back = make_unique<Node>(_value);
        back = &(**back).next;
    }

    T dequeue() {
        if (front == nullptr)
            throw underflow_error("Nothing to dequeue");

        T value = front->value;
        front = move(front->next);

        return value;
    }
};
```

Head or tail first

The ends of a FIFO queue are often referred to as *head* and *tail*. Unfortunately, a controversy exists regarding those terms:

- To many people, items should enter a queue at the tail, and remain in the queue until they reach the head and leave the queue from there. This point of view is justified by analogy with queues of people waiting for some kind of service and parallels the use of *front* and *back* in the above example.
- Other people believe that items enter a queue at the head and leave at the tail, in the manner of food passing through a snake. Queues written in that way appear in places that could be considered authoritative, such as the operating system Linux.

Pipes

In computing environments that support the pipes and filters model for interprocess communication, a FIFO is another name for a named pipe.

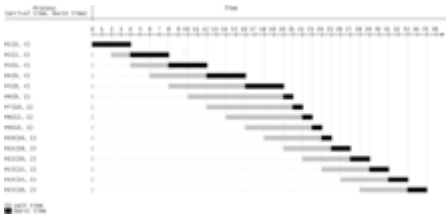
Disk scheduling

Disk controllers can use the FIFO as a disk scheduling algorithm to determine the order in which to service disk I/O requests.

Communications and networking

Communication network bridges, switches and routers used in computer networks use FIFOs to hold data packets en route to their next destination. Typically at least one FIFO structure is used per network connection. Some devices feature multiple FIFOs for simultaneously and independently queuing different types of information.

Electronics


FIFO schedule.

FIFOs are commonly used in electronic circuits for buffering and flow control between hardware and software. In its hardware form, a FIFO primarily consists of a set of read and write pointers, storage and control logic. Storage may be static random access memory (SRAM), flip-flops, latches or any other suitable form of storage. For FIFOs of non-trivial size, a dual-port SRAM is usually used, where one port is dedicated to writing and the other to reading.

A synchronous FIFO is a FIFO where the same clock is used for both reading and writing. An asynchronous FIFO uses different clocks for reading and writing. Asynchronous FIFOs introduce metastability issues. A common implementation of an asynchronous FIFO uses a Gray code (or any unit distance code) for the read and write pointers to ensure reliable flag generation. One further note concerning flag generation is that one must necessarily use pointer arithmetic to generate flags for asynchronous FIFO implementations. Conversely, one may use either a *leaky bucket* approach or pointer arithmetic to generate flags in synchronous FIFO implementations.

Examples of FIFO status flags include: full, empty, almost full, almost empty, etc.

The first known FIFO implemented in electronics was done by Peter Alfke in 1969 at Fairchild Semiconductors.[2] Peter Alfke was later a director at Xilinx.

FIFO full-empty

A hardware FIFO is used for synchronization purposes. It is often implemented as a circular queue, and thus has two pointers:

1. Read pointer / read address register
2. Write pointer / write address register

Read and write addresses are initially both at the first memory location and the FIFO queue is *empty*.

FIFO empty
: When the read address register reaches the write address register, the FIFO triggers the *empty* signal.

FIFO full
: When the write address register reaches the read address register, the FIFO triggers the *full* signal.

In both cases, the read and write addresses end up being equal. To distinguish between the two situations, a simple and robust solution is to add one extra bit for each read and write address which is inverted each time the address wraps. With this set up, the disambiguation conditions are:

1. When the read address register equals the write address register, the FIFO is empty.
2. When the read address LSBs equal the write address LSBs and the extra MSBs are different, the FIFO is full.

See also

- FINO (first in, never out)
- Garbage in, garbage out

References

Citations

1. Kruse, Robert L. (1987) [1984]. *Data Structures & Program Design (second edition)* (https://archive.org/details/datastructurespr0000krus_n1p0/page/150). Joan L. Stone, Kenny Beck, Ed O'Dougherty (production process staff workers) (second (hc) textbook ed.). Englewood Cliffs, New Jersey 07632: Prentice-Hall, Inc. div. of Simon & Schuster. pp. 150 (https://archive.org/details/datastructurespr0000krus_n1p0/page/150). ISBN 0-13-195884-4. "The definition of a finite sequence immediately makes it possible for us to attempt a definition of a list: A 'list' of terms of type T is simply a finite sequence of elements of the set T. ... The only difference among stacks and queues and more general lists is the **operations** by which changes or accesses can be made to the list."
2. Peter Alfke's post at comp.arch.fpga on 19 Jun 1998 (http://www.fpga-faq.com/archives/10775.html#10794)

Sources

- Cummings et al., Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons, SNUG San Jose 2002 (http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf)