Create account Log in

Article Talk Read Edit View history Search

Circular buffer

From Wikipedia, the free encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Deutsch
Español
فارسی
Français
한국어
日本語
Polski
Русский
Српски / srpski
Suomi
Українська
中文
Edit links

This article has multiple issues. Please help **improve it** or discuss these issues on the **talk page**.

- This article's **tone or style may not reflect the encyclopedic tone used on Wikipedia**. *(April 2011)*
- This article **needs additional citations for verification**. *(September 2014)*
- This article or section **appears to contradict itself about the end pointer: it is confused with the next pointer, resulting in a contradiction.** *(April 2013)*

A **circular buffer**, **cyclic buffer** or **ring buffer** is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end. This structure lends itself easily to buffering data streams.

A ring showing, conceptually, a circular buffer. This visually shows that the buffer has no real end and it can loop around the buffer. However, since memory is never physically created as a ring, a linear representation is generally used as is done below.

Contents

Uses [edit]

The useful property of a circular buffer is that it does not need to have its elements shuffled around when one is consumed. (If a non-circular buffer were used then it would be necessary to shift all elements when one is consumed.) In other words, the circular buffer is well-suited as a FIFO buffer while a standard, non-circular buffer is well suited as a LIFO buffer.

Circular buffering makes a good implementation strategy for a queue that has fixed maximum size. Should a maximum size be adopted for a queue, then a circular buffer is a completely ideal implementation; all queue operations are constant time. However, expanding a circular buffer requires shifting memory, which is comparatively costly. For arbitrarily expanding queues, a linked list approach may be preferred instead.

In some situations, overwriting circular buffer can be used, e.g. in multimedia. If the buffer is used as the bounded buffer in the producer-consumer problem then it is probably desired for the producer (e.g., an audio generator) to overwrite old data if the consumer (e.g., the sound card) is unable to momentarily keep up. Also, the LZ77 family of lossless data compression algorithms operates on the assumption that strings seen more recently in a data stream are more likely to occur soon in the stream. Implementations store the most recent data in a circular buffer.

How it works [edit]

A circular buffer first starts empty and of some predefined length. For example, this is a 7-element buffer:

Circular buffer - empty.svg

Assume that a 1 is written into the middle of the buffer (exact starting location does not matter in a circular buffer):

Circular buffer - XX1XXXX.svg

Then assume that two more elements are added — 2 & 3 — which get appended after the 1:

Circular buffer - XX123XX.svg

If two elements are then removed from the buffer, the oldest values inside the buffer are removed. The two elements removed, in this case, are 1 & 2, leaving the buffer with just a 3:

Circular buffer - XXXX3XX.svg

If the buffer has 7 elements then it is completely full:

Circular buffer - 6789345.svg

A consequence of the circular buffer is that when it is full and a subsequent write is performed, then it starts overwriting the oldest data. In this case, two more elements — A & B — are added and they *overwrite* the 3 & 4:

Circular buffer - 6789AB5.svg

Alternatively, the routines that manage the buffer could prevent overwriting the data and return an error or raise an exception. Whether or not data is overwritten is up to the semantics of the buffer routines or the application using the circular buffer.

Finally, if two elements are now removed then what would be returned is **not** 3 & 4 but 5 & 6 because A & B overwrote the 3 & the 4 yielding the buffer with:

Circular buffer - X789ABX.svg

Circular buffer mechanics [edit]

What is not shown in the example above is the mechanics of how the circular buffer is managed.

Start/end pointers (head/tail) [edit]

Generally, a circular buffer requires four pointers:

- one to the actual buffer in memory
- one to the buffer end in memory (or alternately: the size of the buffer)
- one to point to the start of valid data (or alternately: amount of data written to the buffer)
- one to point to the end of valid data (or alternately: amount of data read from the buffer)

Alternatively, a fixed-length buffer with two integers to keep track of indices can be used in languages that do not have pointers.

Taking a couple of examples from above. (While there are numerous ways to label the pointers and exact semantics can vary, this is one way to do it.)

This image shows a partially full buffer:

This image shows a full buffer with two elements having been overwritten:

What to note about the second one is that after each element is overwritten then the start pointer is incremented as well.

Difficulties [edit]

Full / Empty Buffer Distinction [edit]

A small disadvantage of relying on pointers or relative indices of the start and end of data is, that in the case the buffer is entirely full, both pointers point to the same element:[contradiction]

Circular buffer - 6789AB5 full.svg

This is exactly the same situation as when the buffer is empty:

Circular buffer - 6789AB5 empty.svg

To solve this confusion there are a number of solutions:

- Always keep one slot open.
- Use a fill count to distinguish the two cases.
- Use an extra mirroring bit to distinguish the two cases.
- Use read and write counts to get the fill count from.
- Use absolute indices.
- Record last operation.
- Split buffer into two regions.

Always keep one slot open [edit]

This design always keeps one slot unallocated. A full buffer has at most slots. If both pointers refer to the same slot, the buffer is empty. If the end (write) pointer refers to the slot preceding the one referred to by the start (read) pointer, the buffer is full.

The advantage is:

- The solution is simple and robust.

The disadvantages are:

- One slot is lost, so it is a bad compromise when the buffer size is small or the slot is big or is implemented in hardware.
- The test for full requires a modulo operation

Use a fill count [edit]

This approach replaces the end pointer with a counter that tracks the number of readable items in the buffer. This unambiguously indicates when the buffer is empty or full and allows use of all buffer slots.

The performance impact should be negligible, since this approach adds the costs of maintaining the counter and computing the tail slot on writes but eliminates the need to maintain the end pointer and simplifies the fullness test.

The advantage is:

- The test for full/empty is simple

The disadvantages are:

- You need modulo for read and write
- Read and write operation must share the counter field, so it requires synchronization in multi-threaded situation.

Note: When using semaphores in a Producer-consumer model, the semaphores act as a fill count.

Mirroring [edit]

Another solution is to remember the number of times each read and write pointers have wrapped and compare this to distinguish empty and full situations. In fact only the parity of the number of wraps is necessary, so you only need to keep an extra bit. You can see this as if the buffer adds a virtual mirror and the pointers point either to the normal or to the mirrored buffer.

Circular buffer - mirror solution full and empty.svg

It is easy to see above that when the pointers (including the extra msb bit) are equal, the buffer is empty, while if the pointers differ only by the extra msb bit, the buffer is full.

The advantages are:

- The test for full/empty is simple
- No modulo operation is needed
- The source and sink of data can implement independent policies for dealing with a full buffer and overrun while adhering to the rule that only the source of data modifies the write count and only the sink of data modifies the read count. This can result in elegant and robust circular buffer implementations even in multi-threaded environments.[*citation needed*]

The disadvantage is:

- You need one more bit for read and write pointer

Read / Write Counts [edit]

Another solution is to keep counts of the number of items written to and read from the circular buffer. Both counts are stored in unsigned integer variables that overflow and wrap freely, with numerical limits equal or larger than the number of items that can be stored.

The unsigned difference (write_count - read_count) always yields the number of items placed in the buffer and not yet retrieved. This can indicate that the buffer is empty, partially full, completely full (without waste of a storage location) or in a state of overrun.

The advantage is:

- The source and sink of data can implement independent policies for dealing with a full buffer and overrun while adhering to the rule that only the source of data modifies the write count and only the sink of data modifies the read count. This can result in elegant and robust circular buffer implementations even in multi-threaded environments.

The disadvantage is:

- You need two additional variables.

Absolute indices [edit]

It is possible to optimize the previous solution by using indices instead of pointers: indices can store read/write counts instead of the offset from start of the buffer, the separate variables in the above solution are removed and relative indices are obtained on the fly by division modulo the buffer's length.

The advantage is:

- No extra variables are needed.

The disadvantages are:

- Every access needs an additional *modulo* operation.
- If counter wrap is possible, complex logic can be needed if the buffer's length is not a divisor of the counter's capacity.

On binary computers, both of these disadvantages disappear if the buffer's length is a power of two—at the cost of a constraint on possible buffers lengths.

Record last operation [edit]

Another solution is to keep a flag indicating whether the most recent operation was a read or a write. If the two pointers are equal, then the flag will show whether the buffer is full or empty: if the most recent operation was a write, the buffer must be full, and conversely if it was a read, it must be empty.

The advantages are:

- Only a single bit needs to be stored (which may be particularly useful if the algorithm is implemented in hardware)

- The test for full/empty is simple

The disadvantage is:

- You need an extra variable
- Read and write operations must share the flag, so it will probably require synchronization in multi-threaded situation.

Split buffer into two regions [edit]

This approach solves the wrap-around problem by splitting the buffer into a primary region and a secondary region. The secondary region always begins at the buffer start and is not activated until the primary region has reached the buffer's end. Additionally, if the primary region is emptied of data, the secondary region becomes the new primary.

The advantages are:

- The test for full/empty is simple.
- No modulo operation is needed.

The disadvantages are:

- An extra pointer variable is needed.
- Read and write operations are complicated.

Multiple read pointers [edit]

A little bit more complex are multiple read pointers on the same circular buffer. This is useful if you have *n* threads, which are reading from the same buffer, but *one* thread writing to the buffer.

Chunked Buffer [edit]

Much more complex are different chunks of data in the same circular buffer. The writer is not only writing elements to the buffer, it also assigns these elements to chunks[*citation needed*].

The reader should not only be able to read from the buffer, it should also get informed about the chunk borders.

Example: The writer is reading data from small files, writing them into the same circular buffer. The reader is reading the data, but needs to know when and which file is starting at a given position.

Optimization [edit]

A circular-buffer implementation may be optimized by mapping the underlying buffer to two contiguous regions of virtual memory. (Naturally, the underlying buffer‘s length must then equal some multiple of the system’s page size.) Reading from and writing to the circular buffer may then be carried out with greater efficiency by means of direct memory access; those accesses which fall beyond the end of the first virtual-memory region will automatically wrap around to the beginning of the underlying buffer. When the read offset is advanced into the second virtual-memory region, both offsets—read and write—are decremented by the length of the underlying buffer.[1]

Variants [edit]

Perhaps the most common version of the circular buffer uses 8-bit bytes as elements.

Some implementations of the circular buffer use fixed-length elements that are bigger than 8-bit bytes—16-bit integers for audio buffers, 53-byte ATM cells for telecom buffers, etc. Each item is contiguous and has the correct data alignment, so software reading and writing these values can be faster than software that handles non-contiguous and non-aligned values.

Ping-pong buffering can be considered a very specialized circular buffer with exactly two large fixed-length elements.

The Bip Buffer is very similar to a circular buffer, except it always returns contiguous blocks (which can be variable length).[1]

Compressing circular buffers use an alternative indexing strategy, based on elementary number theory, to maintain a fixed-sized, compressed, representation of the entire data sequence [2]

External links [edit]

1. ^ a b Simon Cooke. "The Bip Buffer - The Circular Buffer with a Twist". 2003.
2. ^ John C. Gunther. 2014. Algorithm 938: Compressing circular buffers. ACM Trans. Math. Softw. 40, 2, Article 17 (March 2014) [1]

- CircularBuffer at the Portland Pattern Repository
- Boost: Templated Circular Buffer Container
- http://www.dspguide.com/ch28/2.htm

v · t · e	**Data structures**
Types	Collection · Container
Abstract	Associative array · Double-ended priority queue · Double-ended queue · List · Map · Multimap · Priority queue · Queue · Set (multiset) · Disjoint Sets · Stack
Arrays	Bit array · **Circular buffer** · Dynamic array · Hash table · Hashed array tree · Sparse array
Linked	Association list · Linked list · Skip list · Unrolled linked list · XOR linked list
Trees	B-tree · Binary search tree (AA · AVL · red-black · self-balancing · splay) · Heap (binary · binomial · Fibonacci) · R-tree (R* · R+ · Hilbert) · Trie (Hash tree)
Graphs	Binary decision diagram · Directed acyclic graph · Directed acyclic word graph
	List of data structures

Categories: Computer memory | Arrays