Bencina, R. 2014. Interfacing Real-Time Audio and File I/O. In *Proceedings of the 2014 Australasian Computer Music Conference, ACMC 2014*, Melbourne, Australia. July 9-11.

INTERFACING REAL-TIME AUDIO AND FILE I/O

Ross Bencina
portaudio.com
rossb@audiomulch.com

ABSTRACT

Programming a computer to record or play a sound file in real-time is not as easy as it may seem. The naive approach is to call file I/O APIs from within the routine that handles real-time audio I/O. This leads to audible glitches whenever the time taken to access a file exceeds the time available to deliver a buffer of real-time audio. This paper describes an approach to streaming file playback and recording that operates correctly under these conditions. It performs file I/O in a separate thread, buffers audio data to mask file I/O delays, and uses asynchronous message passing and lock-free queues for inter-thread communication.

1. INTRODUCTION

Real-time audio I/O and file I/O are input/output methods with radically different timing properties. Software that records or plays sound files must perform both types of I/O. To ensure that a constant stream of audio samples is fed to and from the audio hardware, real-time audio programs must adhere to a strict schedule. By contrast, file I/O operates on a best-effort basis, guaranteeing only average-case throughput, not the timeliness of individual operations. These differing timing properties make it difficult to interface between real-time audio and file I/O. Solutions typically involve performing file I/O in a separate thread. This poses further difficulties, as most general-purpose operating systems do not provide real-time-safe inter-thread communication facilities.

This paper presents an efficient and reliable method for implementing real-time file playback and recording. The method comprises streaming algorithms, an asynchronous message passing protocol, a system of lock-free queues, and a "server" thread that performs file I/O. A sample implementation in C++ is also available.[1]

The paper includes the following lock-free mechanisms, which readers may find generally useful for real-time audio programming:

- A light-weight, lock-free message architecture supporting actor-like stream objects. (Aside from the I/O server thread, no message loop, scheduler, or per-thread queues are used.)
- Real-time-safe creation and destruction of stream objects; including non-blocking tear-down when streams are awaiting asynchronous results.
- Light-weight, relaxed-order message queues for returning results to stream objects.

[1] https://github.com/RossBencina/RealTimeFileStreaming

2. CONTEXT

Real-time audio programs must adhere to strict time constraints, yet few non-audio system APIs guarantee timeliness. Here we explore these issues and introduce concepts and terms for later use.

2.1. Real-time audio terminology and time constraints

The operating system triggers a real-time audio program at fixed intervals to consume and/or produce buffers of audio samples. We refer to the user-defined function responsible for producing and/or consuming audio data as the **real-time audio routine**. The period of time between successive invocations of this routine is referred to as the **buffer period**. "Low-latency" interactive audio applications, such as music production and performance, ideally employ buffer periods between one and six milliseconds.

To ensure that real-time audio does not glitch, the time taken to execute the real-time audio routine must always be less than the buffer period. As a consequence, all operations performed by the real-time audio routine must be guaranteed to complete in short and bounded time. Informally, we refer to operations with this property as *real-time-safe*.

2.2. Library and system calls are not real-time-safe

General-purpose operating systems do not guarantee worst-case time bounds for system calls. There are many causes of unbounded time behaviour in such systems, including: algorithms with poor worst-case complexity, code that waits for hardware, code that performs blocking interactions with other threads, the thread scheduler and/or the virtual memory paging mechanism.

In particular, memory allocation and unconditional locks (mutexes) are not real-time-safe. Memory allocators often employ algorithms with poor worst-case time bounds. Locks may be subject to unbounded priority inversion.[2]

Furthermore, we can extrapolate that most system APIs are not real-time-safe because they directly allocate memory or use locks, or depend on code that does. An inevitable conclusion follows that any real-time audio routine must avoid calling system APIs. In fact, this is a stated requirement of many real-time audio I/O APIs.

[2] Of Windows, OS X and Linux, only Linux offers the option of real-time-safe mutexes. Windows uses a probabilistic priority inversion avoidance mechanism with no time guarantees. Apple's libc mutex implementation contains the comment "TODO: Priority inheritance stuff." Even on Linux, there is no guarantee that third-party code uses PTHREAD_PRIO_INHERIT.

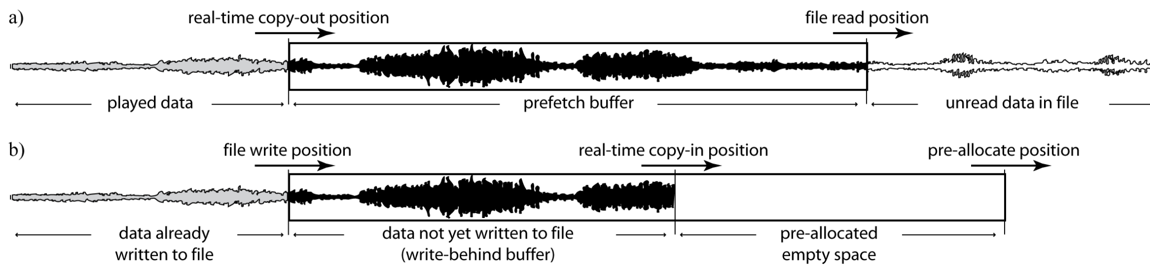

Figure 1: a) Playback buffering scheme, b) Record buffering scheme.

2.3. File I/O is not real-time-safe

File operations–such as opening, closing, reading and writing–take considerable time due to delays inherent to consumer-grade storage devices (hard disks, SSDs). Consider the following:

- Fast spinning disks exhibit seek times of the order of 5 to 7 milliseconds for a single I/O operation.
- SSDs are typically specified to complete 99% of operations in less than one millisecond, but do not guarantee worst-case time for all operations.
- Storage devices are a shared resource. When multiple tasks try to perform I/O simultaneously, operations will be queued and delayed as a consequence.
- Blocking file I/O functions such as `fread()` may wait for data to be read from the storage medium despite the use of anticipatory read-ahead by the operating system.

These considerations are amplified when multiple files are streamed simultaneously, as in a multi-track digital audio workstation.

3. STREAMING PLAYBACK AND RECORD

This section describes, in abstract, two separate processes that may be executed from a real-time audio routine: playing audio from a file and recording audio to a file. We assume a model where the real-time audio routine “pulls” data from the file during playback, and “pushes” data to the file during recording.

3.1. Playback: asynchronous reads and read-ahead buffering

Playback of audio data from a file involves a repeated process of reading audio data from the file and copying the data into an audio output buffer. For this process to be successful, the audio routine must feed audio data to the operating system within a prescribed time frame. Any delay will lead to audible gaps or glitches.

When compared to the time available to run a low-latency real-time audio routine, file I/O operations can take a long time to execute. To avoid stalling the audio routine it is necessary to perform I/O *asynchronously*. This entails splitting I/O operations into two phases: (1) initiate a request to perform an operation (e.g. read data from a file), (2) at a later time, I/O completes and the file data becomes available. Observe that a blocking I/O operation has been converted into two non-blocking operations (*initiate* and *complete*), which are assumed to be real-time-safe.

While asynchronous I/O avoids stalling the audio routine, the potential delay between I/O initiation and completion leaves open the problem of delivering an uninterrupted stream of audio data.

To achieve uninterrupted playback, the audio routine must continuously *prefetch* (read-ahead) audio data from the file into a buffer. In the event that an asynchronous file read operation is delayed, there must be sufficient audio data in the buffer to support uninterrupted playback. Figure 1a illustrates the process.

During streaming playback the real-time audio routine is responsible for two tasks: (1) perform read-ahead by issuing asynchronous file read requests to keep a prefetch buffer full; and (2) copy audio data from the prefetch buffer to the system's audio output buffer. Initially, the real-time audio routine outputs silence until the prefetch buffer becomes full.

3.2. Recording: prefetching buffer space, asynchronous writes and write-behind buffering

Recording audio data to a file involves copying the audio data from the system audio buffer to an intermediate buffer, then writing the buffered data to the file.

As with playback, file writes must be performed asynchronously to avoid stalling the real-time audio routine. The buffering scheme differs from the playback case in that the recording audio routine must always pre-allocate sufficient buffer space to capture incoming audio data. Once filled, the buffer space is passed off to the asynchronous file write operation, which writes the data to file (a process known as *write-behind*). When the write operation has completed, the filled buffer space can be reclaimed or reused. See Figure 1b.

The length of the pre-allocated space must be sufficient to mask the worst-case time taken to allocate more space. The write-behind buffer needs to be long enough to mask the worst-case delay for file write operations.

4. MESSAGE PROTOCOL AND ALGORITHMS

Our implementation strategy uses an “I/O server” thread to perform memory allocation and file I/O. Clients, such as the real-time audio routine, communicate with the I/O server using asynchronous messages. In this section we describe the message protocol for asynchronous file I/O, the streaming algorithms expressed in terms of the protocol, and requirements for a messaging infrastructure suitable for real-time use. In particular we require the availability of a real-time-safe mechanism for sending requests and receiving replies.

4.1. An asynchronous message protocol for file I/O

The asynchronous protocol comprises messages (requests) sent by a client to the I/O server and corresponding results sent by the server in reply. From the client's perspective, *I/O invocation* maps to sending a request and *I/O completion* maps to receiving a reply.

The notation used below reads as follows: **MSG a → b** denotes message **MSG** with parameter **a** sent asynchronously to the server and result/reply **b** later received asynchronously by the client. **MSG a → ∘** is similar except there is no reply and the operation is assumed to succeed. **(a,b,...)** denotes a collection of parameters or results. **a | b** denotes that either result **a** or result **b** is returned.

4.1.1. Opening and closing file handles

OPEN_FILE (path, accessMode) → fileHandle | ***error***

Given a file path (a string) and an access mode (read-only, read-write, etc.), open the file and return a file handle or an error code (if an error occurs). The file handle is an opaque identifier used by the client to refer to the file when requesting I/O.

CLOSE_FILE fileHandle → ∘

At some point after receiving a valid file handle, the client must close the handle with a CLOSE_FILE message.

4.1.2. Data blocks

Read and write operations deal in pointers to server-allocated data blocks that hold regions of file data. All of a file's data blocks contain the same number of audio samples. Data blocks are addressed by their starting position within the file. In addition to the raw file data, each data block maintains book-keeping fields; such as an indication of which areas of the block contain valid data.

4.1.3. Read operations

READ_BLOCK (fileHandle, position)
→ dataBlock | ***error***

Given a file handle and a position in the file, allocate a data block, read data from the file into the block, store the valid data range, and return the data block or an error code (if an error occurred).

RELEASE_READ_BLOCK (fileHandle, dataBlock) → ∘

Data blocks acquired with READ_BLOCK must be released using the RELEASE_READ_BLOCK message, which causes the data block to be deallocated.

4.1.4. Write operations

ALLOCATE_WRITE_BLOCK (fileHandle, position)
→ dataBlock | ***error***

Given a file handle and a position in the file, allocate the data block in which the client will write data for a specified region of a file. If the specified region is a valid area of the file, fill the data block with data from the file (similar to read), otherwise leave the data area uninitialised and mark it as invalid. Return the data block or an error code (if an error occurs).

After acquiring a block with ALLOCATE_WRITE_BLOCK, the client must either commit or release the block using either COMMIT_MODIFIED_WRITE_BLOCK or RELEASE_UNMODIFIED_WRITE_BLOCK.

COMMIT_MODIFIED_WRITE_BLOCK (fileHandle, dataBlock)
→ ∘

COMMIT_MODIFIED_WRITE_BLOCK writes the block's data to the file and releases the data block.

RELEASE_UNMODIFIED_WRITE_BLOCK
(fileHandle, dataBlock) → ∘

RELEASE_UNMODIFIED_WRITE_BLOCK deallocates the data block without writing any data to the file. This message should only be used if the block was not modified.

4.1.5. Message ordering constraints

A client may only issue READ_BLOCK or ALLOCATE_WRITE_BLOCK requests for a particular file handle prior to sending the CLOSE_FILE message. This requires the server to process requests in first-in first-out (FIFO) order, otherwise the server might encounter a block request after having closed the file. COMMIT_MODIFIED_WRITE_BLOCK requests that extend the file length also require that requests be processed in order, unless the file system supports files with "holes."

The client may RELEASE or COMMIT acquired data blocks after sending the CLOSE_FILE message. The I/O server only closes the native file handle once all data blocks have been released or committed.

4.2. Streaming algorithms

We now present the playback and recording algorithms. The algorithms assume that a file handle has been obtained by sending the OPEN_FILE message and receiving the resulting file handle.

Algorithm: streaming file playback. Maintain a prefetch queue referencing N data blocks ordered by ascending file position. Maintain a playback index referring to the next audio sample to output. At each time-step, output a single audio sample. (See Figure 2a.)

1. [Setup.] Set the playback index to zero. Issue N READ_BLOCK requests for sequential data blocks. Order the block references in the prefetch queue, marking all blocks as *pending* until they have been received. Output silence until the replies to all N READ_BLOCK requests have been received. At each subsequent audio time-step perform the following three steps:
2. [Retiring replies.] Whenever a READ_BLOCK reply is received, mark the block as *received* or as *error* if an error result is received.
3. [Output an audio sample.] Output (copy out) the sample referred to by the playback index–this indexes into the front-most data block in the prefetch queue. Output silence if the block is *pending* or if an *error* is indicated.
4. [Advance playback index, maintain the prefetch queue.] Increment the playback index. If the index now points into the second block in the queue: (1) pop the first block from the queue and issue a RELEASE_READ_BLOCK request for that block; (2) issue a READ_BLOCK request for the block following the latest block requested so far. ∎

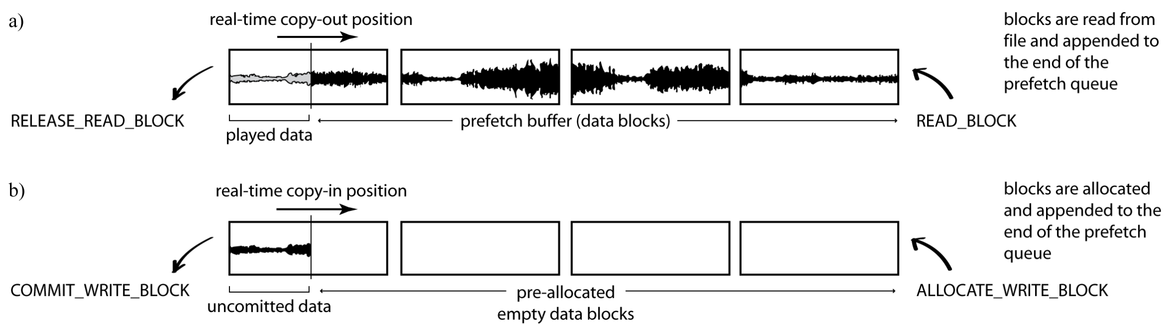

Figure 2: a) A prefetch queue of data blocks for playback, b) A prefetch queue of data blocks for recording

The algorithm for recording is similar to the above, except that data is copied in to the blocks rather than out of them (see Figure 2b). The messages are changed as follows: `READ_BLOCK` becomes `ALLOCATE_WRITE_BLOCK`. When data is copied into a block the block is marked as *modified.* Modified blocks are released using `COMMIT_WRITE_BLOCK`. Unmodified blocks are released using `RELEASE_UNMODIFIED_WRITE_BLOCK` (e.g. to flush the prefetch queue when recording stops).

4.3. Implementation requirements and desiderata

Any implementation of the messaging protocol must fulfil the following requirements:

- Allocating, deallocating, sending requests and receiving replies must be real-time-safe.
- The protocol requires the server to process events in FIFO order. Clients may receive replies in any order.

In addition, the following desiderata would contribute to the usability and flexibility of a solution:

- Support immediate disposal of streaming state, without having to wait for pending replies.
- Construction and tear-down of streaming state should be real-time-safe.
- Support creation and destruction of streams in any thread, and allow stream states to migrate between threads (e.g. create a stream in one thread and use it in another, or use a stream in different threads at different times). This implies being able to send requests and receive replies from any thread.
- Do not use thread-local storage or client-managed inter-thread queueing infrastructure.

The first point is desirable when combining asynchronous streams with imperative code. For instance it is desirable to be able to embed a stream in a C++ object, and to delete the object without having to wait for the stream to receive and process pending asynchronous replies.

The last three points express a desire for programming flexibility and minimising maintenance overhead. They seek to avoid the inconvenience of using rigid topologies of queues and threads, and/or requiring certain operations to be performed in certain threads.

Stream operations are not required to be thread-safe or re-entrant. Stream state may be passed between threads but not simultaneously accessed from multiple threads.

5. IMPLEMENTATION IN C++

Figure 3 shows the multi-threaded communication structure of the solution. The I/O Server thread has a queue (mailbox) for inbound requests. Client streams send requests to the server by enqueueing Request objects into the server's mailbox.

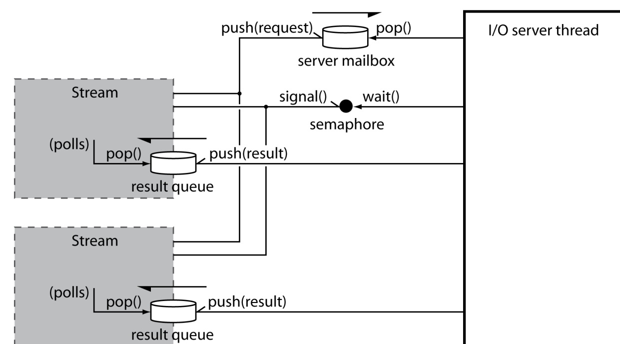

Figure 3. Multi-threaded communication structure showing two client streams.

Streams are not associated with a particular thread and may move freely between threads. In contrast to architectures that employ an incoming message queue for each thread, we use a separate result queue for each client stream. When the server completes a request, it posts the reply (if any) into the result queue specified by the request. Client streams poll their result queues as part of the playback or recording process. There is no requirement for clients to be signalled by the server.

5.1. Requests and message queues

We now turn our attention to the implementation of the Request objects and lock-free message queues.

5.1.1. Request objects

A single fixed-size Request data type (a C/C++ `struct`) is used to represent all message protocol requests and replies. Replies are returned to the client using the same Request instance that was used to initiate the request. The Request data type includes a tag indicating the specific request type (`OPEN_FILE`, `CLOSE_FILE`, etc.), a union containing the parameters for each request type, and fields common to all requests (link pointers, result code, a pointer to a client queue to return replies/results, and fields reserved for client-only use).

Each Request contains two “next” pointer fields that allow the Request to be simultaneously linked into two

separate linked structures or queues. One next pointer is used for linking the request into client-server communication data structures, for example to enqueue the Request to the server, or to enqueue a reply to the client. A second client-only next pointer is used by the client to link Requests together into client-local data structures.

With the exception of client-only fields, the client should not modify a Request object from the time that it is sent to the server until its reply is received. When Request objects are not queued with the server, all Request fields are available for client use.

A global lock-free freelist of Request objects supports allocation and deallocation from any thread. The freelist is implemented using the "IBM Freelist" algorithm (IBM 1983; Treiber 1987; Michael and Scott 1998), a last-in first-out (LIFO) stack that supports non-blocking push and pop operations.

5.1.2. A lock-free "pop-all" LIFO stack

Listing 1 shows a lock-free LIFO stack algorithm that supports *push, is-empty* and *pop-all* operations.[1] It supports concurrent operations from multiple threads. We use this algorithm as the basis of our message queues.

```
struct Node { Node *next; };
struct Stack { Node *top; };
void init(Stack& s) { s.top = NULL; }
void push(Stack& s, Node *n, bool& wasEmpty) {
  do {
    Node *top = s.top;
    n->next = top;
    wasEmpty = (top==NULL);
    // CAS: atomic compare-and-swap
    // set s.top to n only if s.top == top
  } while(!CAS(&s.top, top, n));
}
bool is_empty(Stack& s) { return (s.top==NULL); }
Node *pop_all(Stack& s) {
  if (s.top==NULL) return; // don't modify if empty
  // XCHG: atomic exchange
  // set s.top to NULL, return old s.top
  return XCHG(&s.top,NULL);
}
```

Listing 1: "Pop-all" concurrent LIFO stack algorithm.

The algorithm works as follows: `s.top` points to the top of a LIFO stack of linked nodes terminated by a `NULL` value. The push algorithm (identical to the IBM freelist) attempts to link a node onto the top of the stack, and retries if a conflict with a concurrent operation is detected. The pop-all algorithm uses an atomic exchange operation to replace `top` with `NULL` and returns the previous contents of the stack–a `NULL`-terminated linked list in LIFO order.[2]

5.1.3. Sending requests to the server

In section 4.3 we established that messages should be received and processed by the I/O server in FIFO order, and that it is desirable to support posting requests to the server from arbitrary threads. To achieve these goals we use a multiple-producer single-consumer (MPSC) FIFO queue. A simple algorithm with this property is the so-called "reversed IBM freelist."

The algorithm can be described in terms of the pop-all LIFO stack from the previous section: Producers enqueue requests by pushing them onto the pop-all stack. The consumer maintains a separate consumer-local stack in FIFO order. Requests are dequeued from the consumer-local stack. When the consumer-local stack is empty, the consumer checks whether the pop-all stack is empty, if not it pops *all* items from the pop-all stack and reverses their order into the consumer-local FIFO queue, from whence further requests are dequeued. The result is a very simple lock-free MPSC FIFO queue.

When a client enqueues a new request it signals the server using a semaphore (or on Windows, an auto-reset Event Object). Since the server only waits on the semaphore when its lock-free stack is empty, the client need only signal the semaphore when it pushes a request onto an empty stack. The pop-all stack's push operation indicates when it has pushed onto an empty stack.

5.1.4. Receiving results from the server: result queues

Each stream has its own **result queue**, an object comprising a lock-free queue and a book-keeping counter. Results are only enqueued onto result queues by the I/O server, and dequeued (polled) by the stream that owns the result queue. The streaming algorithms do not require replies to be returned in order, so the queue does not need to guarantee delivery order. We use a lock-free single-producer single-consumer (SPSC) relaxed-order queue. It can be implemented similarly to the reversed queue from the previous section, with reversing omitted.

A result queue maintains a count of expected results. The client increments the count when it sends a request that expects a reply, and decrements the count when the result is received. When the count drops to zero no further results are expected. (Note that this scheme precludes requests with optional replies.)

We embed result queues directly in Request objects. Result queues only use one or two words of storage, and hence can be stored in the request-type-specific parameter area of a Request. This lets us allocate result queues using the real-time-safe global Request freelist and to send and receive them as messages. The latter feature is used by the result queue clean-up process described later.

5.1.5. The server process

The server waits (blocks) on a semaphore until new requests are posted to its queue. Received requests are processed sequentially in FIFO order.

In order to manage the lifetime of native file handles, the server associates a reference count with each open file. The count is incremented when the server returns a file handle to the client in response to `OPEN_FILE`, and whenever the server returns a block to the client. The count is decremented on `CLOSE_FILE` and whenever a block is received from the client. The native file handle is closed when the reference count drops to zero.

---

[1] The presentation here lacks memory fences and atomic loads and stores. Check the example code for a more complete implementation.

[2] Although of uncertain origin, this algorithm, as well as the reversing method discussed next, is well known to those familiar with the art of lock-free programming, see e.g.: https://groups.google.com/d/msg/lock-free/i0eE2-A7eIA/g745KEEx2JIJ

5.2. Client streams

We now present the client stream data structure and its life cycle: creation, maintenance, and destruction.

5.2.1. Client stream data structures

All client stream data structures are built by linking together Request objects. Request object allocation and deallocation is real-time-safe. So too is allocation and deallocation of stream data structures.

A fully constructed stream is illustrated in Figure 4. A stream comprises a result queue Request (for receiving results from the server), an `OPEN_FILE` Request (which is transformed to `CLOSE_FILE` when closing the stream), and zero or more block Requests linked into a tail-queue that serves as the stream's prefetch buffer. A stream is referenced by client code as an opaque pointer (handle). Internally the pointer points to the result queue.

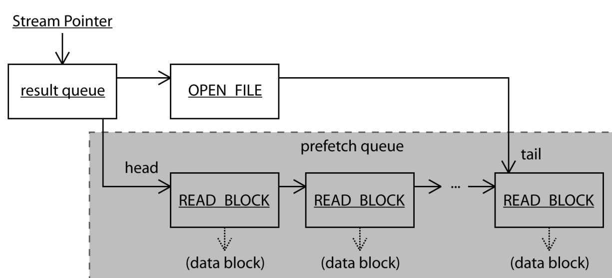

Figure 4: A fully constructed stream using linked Request objects. Each solid rectangle is a Request.

In addition to the result queue, `OPEN_FILE` Request and prefetch queue, a stream maintains the following state variables:

- The overall state of the stream; one of: {OPENING, OPEN_IDLE, OPEN_BUFFERING, OPEN_STREAMING, ERROR}.
- A count of pending blocks (those that have been requested but not yet returned to the stream).
- An index that tracks how much data has been copied to or from the front-most data block.

These state variables are stored in unused and client-use Request fields.

5.2.2. Creating and opening a stream

To create and open a stream, the client first creates two Request objects: a result queue and an `OPEN_FILE` request. The `OPEN_FILE` request is sent to the server, specifying the result queue as its return address. The stream state is set to OPENING, and an opaque stream pointer that points to the result queue is returned to the client. Some time later the `OPEN_FILE` request is returned in the result queue. When the stream gains control it dequeues the result and handles it as follows:

- If the `OPEN_FILE` request was successful, it is linked to the result queue. It contains the file handle that will be used for all block requests. The stream state is set to OPEN_BUFFERING and the prefetch queue pointers are cleared to indicate that the prefetch queue is empty.
- If the `OPEN_FILE` Request failed, the error result is copied to the result queue's error field, the stream state is set to ERROR, and the Request object is deallocated.

5.2.3. Maintaining the prefetch queue

The prefetch queue is represented using a singly-linked tail queue. A tail queue affords constant-time insertion at the back and constant-time removal from the front. The queue is formed from pending and completed `READ_BLOCK` or `ALLOCATE_WRITE_BLOCK` requests (see Figure 4).

The reading or writing process works by copying data from or to the front-most block in the prefetch queue. A numeric field in the front-most block request is used to track how much data has been copied so far. Once the process has finished with a block, the block is returned to the server and a new block is requested and inserted at the back of the queue. Block requests are linked into the queue as soon as they are created.

The block request's request type field is also used (overloaded) to track the state of each data block. The block state is one of: {PENDING, READY, MODIFIED, ERROR}. The request type (`READ_BLOCK` or `ALLOCATE_WRITE_BLOCK`) denotes the PENDING state. When a request's reply is received from the server, the block's state is set to READY or ERROR. The MODIFIED state is used to determine whether a recording stream should `COMMIT` or `RELEASE` the block.

In its simplest form, *seeking* clears the prefetch queue, returns all acquired blocks to the server, and issues `READ` or `ALLOCATE` block requests starting at the new seek position. Care must be taken with PENDING requests since they are yet to arrive in the result queue. They can be removed from the prefetch queue if flagged "return to the server on arrival."

Operations on the prefetch queue also affect the stream's global state. If the streaming process finds that the front-most block is PENDING, it signals an underflow condition and causes the stream state to be set to OPEN_BUFFERING. The stream state is set back to OPEN_STREAMING when all pending blocks have been received (this condition is detected using the pending blocks count). The BUFFERING and STREAMING states support behaviour that suspends playback while the stream is buffering. If the prefetch buffer is correctly sized, the BUFFERING state should only be encountered immediately after seeking.

5.2.4. Destroying a stream

To destroy a stream, all resources associated with the stream must be released. The destruction process needs to account for the case where Requests are in flight when the client instigates stream destruction. For example, the stream may be waiting for the reply to an `OPEN_FILE` request or to one or more `READ_BLOCK` requests. In these cases the result queue can not be destroyed until all results have been received. Then once received, results must be cleaned up: a file handle returned by `OPEN_FILE` must be closed, blocks acquired by `READ_BLOCK` or `ALLOCATE_WRITE_BLOCK` must be released.

The stream destruction algorithm makes use of three facts: (1) each request that returns a resource has a deterministic clean-up procedure, (2) the result queue can be sent as a message to the server, and (3) the result queue includes a counter indicating the number of replies that it is expecting. The counter makes it possible to determine when the result queue has received all replies, and hence when it can be safely destroyed.

Destruction of a stream that is in any of the OPEN_... states involves the following algorithm:

Algorithm: destroying an open stream.

1. [Handle block requests in the prefetch queue.] For each block request in the stream's prefetch queue: If the block request is READY or MODIFIED, transform the block request into a RELEASE or COMMIT message respectively, and send the request to the server (the server will release or commit the block). Discard references to PENDING block requests, they have yet to arrive in the result queue and will be cleaned up later. Deallocate requests in the ERROR state directly to the global freelist.
2. [Close the open file handle.] Transform the OPEN_FILE request into a CLOSE_FILE and send it to the server.
3. [Dispose the result queue.] If the result queue's expected result count is zero, deallocate the result queue's containing Request object to the global freelist. Otherwise, send the result queue to the server using a newly defined message type: **CLEANUP_RESULT_QUEUE resultQueue → ◦**
   This message delegates responsibility for cleaning up any pending requests to the server (see next section). ∎

Destruction of a stream in the OPENING state sends the result queue request to the server as a CLEANUP_RESULT_QUEUE request. Destruction of a stream in the ERROR state deallocates the result queue Request object to the global freelist.

5.2.5. *Server processing of* CLEANUP_RESULT_QUEUE

When the server receives a CLEANUP_RESULT_QUEUE request it pops any queued results from the result queue. For each popped result it decrements the result queue's expected result count and performs the result's compensating operation (e.g. OPEN_FILE transforms to CLOSE_FILE, READ_BLOCK transforms to RELEASE_READ_BLOCK). If the result queue's expected result count is zero, the result queue is deallocated. Otherwise, the result queue is marked with an *awaiting clean-up* flag. Subsequently, whenever the server completes a request, it checks the result queue's flag. If the flag is set, rather than enqueueing the reply, the server decrements the expected result count and performs the compensating operation. Once the expected result count drops to zero the result queue is deallocated to the global freelist.

The result queue clean-up process ensures that all resources associated with a stream are released. It also provides a non-blocking real-time-safe mechanism for destroying streams. As the result queue is itself a Request object, the CLEANUP_RESULT_QUEUE message can always be issued. It cannot fail due to failure to allocate a Request.

6. DISCUSSION AND FUTURE WORK

The presented algorithms for real-time stream operations have time complexity of either O(1) or O(N) in the length of the prefetch queue. Strategies can be devised to reduce the cost to O(1) by moving responsibility for prefetch queue creation and tear-down to the I/O server.

Ideally the protocol would not require Request objects to be allocated or deallocated while the stream is running. Request allocation and transport overhead can be reduced by altering the protocol to allow requests to be reused. Methods to achieve this include: always return results, provide messages that perform multiple operations (e.g. read N blocks), and combine messages (e.g. combine RELEASE_BLOCK and READ_BLOCK into a single Request).

The protocol described in section 4 does not communicate the file length or audio data format information to the client stream. If needed, this information can be communicated as additional results from OPEN_FILE.

As presented, the method does not reliably support multiple real-time streams. The seek process injects N consecutive READ_BLOCK operations into the I/O server queue. These operations may delay more urgent operations that are enqueued later. To address this, assign a deadline to each request and have the server perform operations in earliest-deadline-first (EDF) order.

The performance of the lock-free queues used here are sufficient for the expected contention rate. In a high-throughput scenario other lock-free and wait-free queue algorithms should be considered.

Some messages involve shared access to the Request object by both client and server. This does not result in any data races since access is limited to disjoint fields in each thread. It does however introduce the possibility of false-sharing. It would be interesting to investigate the impact of false-sharing on protocol performance.

We have omitted discussion of handling formatted sound files (.wav, .aiff, .mp3, etc.). One way to handle formatted sound files is for the I/O server thread to perform formatted I/O using an existing sound file I/O library. Alternatively, the I/O server could be extended to implement sound file container parsing and audio format conversion.

This paper has presented a simplified model of a real-time file streaming system currently under development by the author. The system under development is designed to support multi-threaded access to streaming file I/O for a multi-core-capable real-time audio engine. Beyond what has been presented here, the system supports caching and sharing file handles and data blocks amongst multiple client streams. Requests may be prioritised, reprioritised and cancelled. Multiple native I/O operations may be queued to the operating system concurrently. The system supports parsing sound file containers and transparent data format conversion. All of this is achieved using an asynchronous messaging model similar to the one presented here.

7. RELATED WORK

I/O has been an important aspect of computing from the beginning. Knuth (1997) surveys early work and describes I/O using linked chains of buffer descriptors.

Operating system kernels use queues of linked I/O requests (Comer 2011). Windows NT's *I/O request packets* (Russinovich et al. 2012) have their origins dating back at least to DEC's RSX11 operating system (Pellegrini and Cutler 1974). The Parameter Block API of Mac OS Classic (Apple 1985) is an asynchronous file I/O API that entails enqueueing parameter blocks (requests) to be processed asynchronously by the operating system.

Fixed-size I/O data blocks are commonly used for device-independent kernel interfaces (Tanenbaum 2001). Pai and colleagues (2000) conducted an extensive study of models for exchanging I/O buffers. Brustoloni (1997) provides a useful taxonomy of buffer sharing models.

The solution described here is an instance of the "Half-Sync/Half Async" design pattern (Schmidt and Cranor 1995), with the variation that clients do not block if data is unavailable. Asynchronous message queueing and fixed size allocation pools are standard techniques in real-time systems development (Douglass 2003).

For a gentle introduction to lock-free algorithms, see (Michael 2013). The *Synthesis* operating system kernel is notable for its use of lock-free data structures for queueing (Massalin and Pu 1992). The lock-free pop-all LIFO and reverse IBM freelist described in this paper, were encountered on the comp.programming.threads newsgroup during 2005-2008. Participants included Joe Seigh, Chris Thomasson and Dmitry Vyukov.

Lock-free ring buffers are used for transferring audio data and messages in real-time audio applications, primarily as a means to avoid priority inversion, as in for example PortAudio (Bencina and Burk 2001). Lock-free techniques for computer music systems were discussed by Fober and colleagues (2002). A concrete example of use is James McCartney's SuperCollider 3 synthesis server, scsynth (2002). See (Bencina 2011) for an analysis of the message passing techniques in scsynth. Another application of lock-free techniques in computer music is Shelton's real-time live coding environment (2011).

Anderson and colleagues (1997) describe a real-time video conferencing application using a scheme of lock-free queues for inter-thread communication. That publication is notable for its analysis of the hard-real-time-safety of lock-free queues on a uniprocessor. A related analysis is given by (Cho 2006). The author is not aware of similar analysis for multi-processor systems.

8. ACKNOWLEDGEMENTS

Many thanks to Tony Holzner, Scott Brewer, Phil Burk, Andrew Bencina and the anonymous reviewers for their helpful feedback on earlier drafts of this paper.

9. REFERENCES

Anderson, J. H. et al. 1997. "Real-Time Computing with Lock-Free Shared Objects." *ACM Transactions on Computer Systems.* 15(2):134–165.

Apple Computer, Inc. 1985. *Inside Macintosh, Volume II.* Reading, Massachusetts: Addison-Wesley, pp. II-97 – II-119.

Bencina, R. and P. Burk. 2001. PortAudio – an open source cross platform audio API. In *Proceedings of the 2001 International Computer Music Conference*.

Bencina, R. 2011. "Inside scsynth," In Wilson, Cottle, Collins eds. *The Super Collider Book,* Massachusetts: The MIT Press, pp. 721–740.

Brustoloni, J. C. 1997. "Effects of Data Passing Semantics and Operating System Structure on Network I/O Performance." Doctoral thesis. Carnegie-Mellon University Pittsburgh PA, School of Computer Science.

Cho, H. et al. 2006. "Lock-Free Synchronization for Dynamic Embedded Real-Time Systems." In proceedings *Design, Automation and Test in Europe, DATE '06.*

Comer, D. 2011. *Operating System Design - The Xinu Approach, Linksys Version.* CRC Press, pp. 373–381.

Douglass, B. P. 2003 *Real Time Design Patterns.* Boston: Addison-Wesley, sections 5.3 and 6.3.

Fober, D. et al. 2002. "Lock-Free Techniques for Concurrent Access to Shared Objects." *Actes des Journ'ees d'Informatique Musicale JIM2002*, Marseille, pp.143–150.

International Business Machines Corporation (IBM). 1983. *IBM System/370 Extended Architecture, Principles of Operation. First Edition,* pp. A-44 – A-45.

Knuth, D. E. 1997. *The Art of Computer Programming, Volume 1.* 3rd Ed. Upper Saddle River, NJ: Addison-Wesley, pp. 215–231.

Massalin, H. and C. Pu. 1992. "A Lock-Free Multiprocessor OS Kernel." ACM SIGOPS Operating Systems Review, 26(2):108.

McCartney, J. 2002. "Rethinking the computer music language: SuperCollider." *Computer Music Journal* 26(4) 61–68.

Michael, M. M. and M. L. Scott. 1998. "Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors." *Journal of Parallel and Distributed Computing* 51(1):1–26.

Michael, M. M. 2013. "The Balancing Act of Choosing Nonblocking Features." *ACM Queue* 11(7).

Pai, S. et al. 2000. "IO-Lite: A Unified I/O Buffering and Caching System." *ACM Transactions on Computer Systems.* 18(1):37–66.

Pellegrini, M. and Cutler, D. 1974. "RSX-11M Working Design Document," Digital Equipment Corp, Maynard Mass, p. 35.

Russinovich, M., et al. 2012. *Windows Internals.* 6th ed. Part 2. Redmond, Washington: Microsoft Press, p. 28.

Schmidt, D. C. and Cranor, C. D. 1995. "Half-Sync/Half-Async - An Architectural Pattern for Efficient and Well-structured Concurrent I/O." In *Proceedings of the 2nd Annual Conference on the Pattern Languages of Programs.*

Shelton, R. J. 2011. "A Lock-Free Environment for Computer Music: Concurrent Components for Computer Supported Cooperative Work." PhD thesis. The University of Melbourne, Department of Computer Science and Software Engineering.

Tanenbaum, A. S. 2001. "Modern Operating Systems," 2nd Ed. 2001, New Jersey: Prentice Hall. p. 298.

Treiber, R. K. 1986. "Systems Programming: Coping with Parallelism". Technical Report RJ 5118, IBM Almaden Research Center.