INTERFACING REAL-TIME AUDIO AND FILE I/O

Ross Bencina

portaudio.com
rossbencina.com
rossb@audiomulch.com
@RossBencina

Example source code:
https://github.com/RossBencina/RealTimeFileStreaming

Q: How to stream audio data to/from file in real-time without glitching?

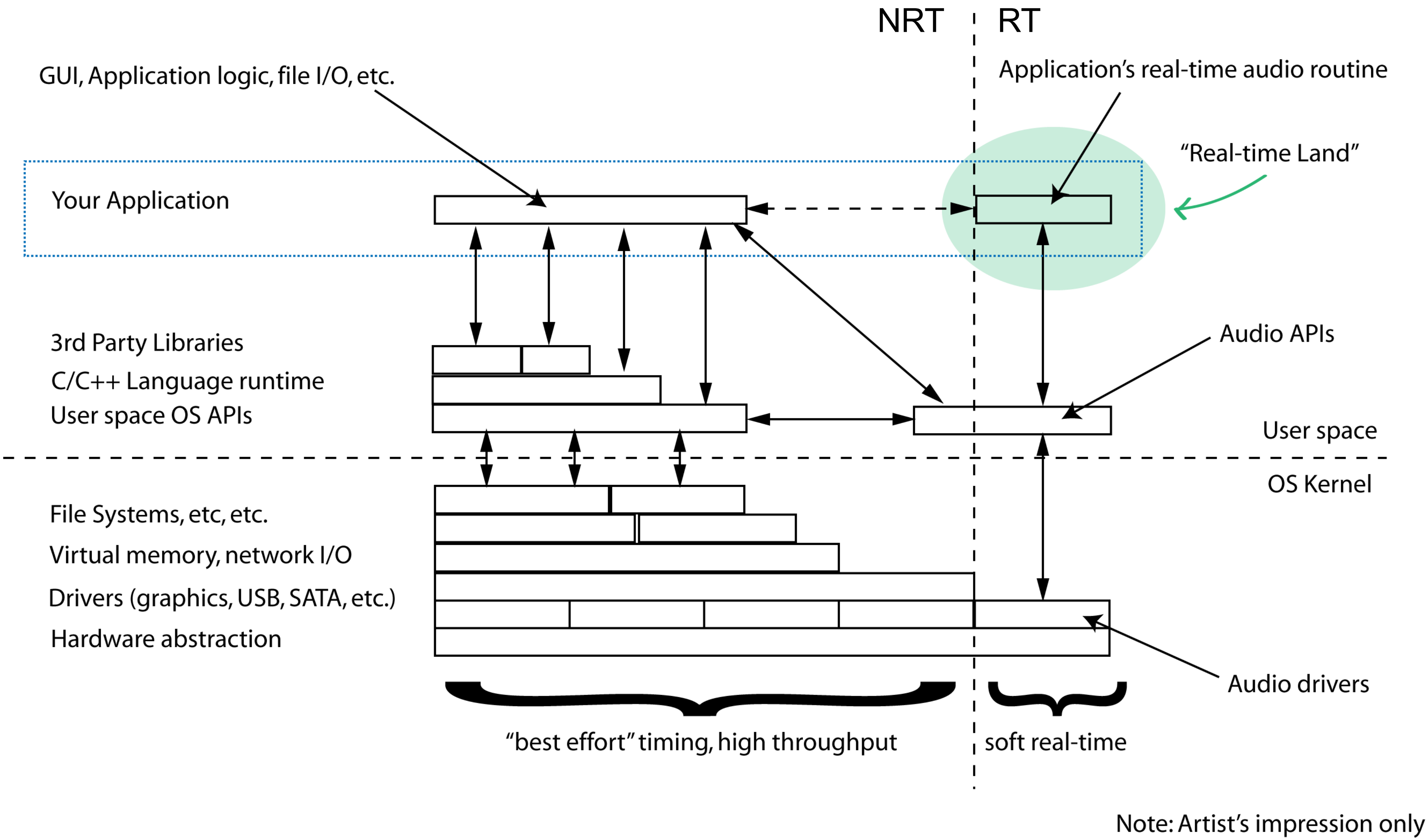
Q: How to stream audio data to/from file in real-time without glitching?

Two problems:

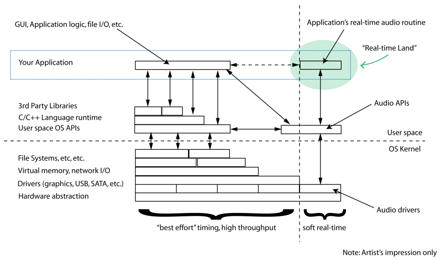
1. Real-time audio programming

2. Unavoidable file access delays

Real-time audio programming context



See also:
http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothing



"Real-Time Land"

Time constrained: must meet deadlines

Write code with deterministic time properties ("real-time-safe")

Limited programming model:

- Can't allocate memory (by normal means)
- Can't use locks (c.f. priority inversion)
- Can't call system APIs (in general, see above)

Unavoidable file access delays

File access takes an unpredictable amount of time and is not real-time safe

Buffer management, hardware access queueing, and performing I/O all take time.

Ballpark numbers:
~? network file access (pick a number)
~10ms+ for mechanical hard disks
~50us for SSD, plus ??? jitter, GC

Interlocking Concerns:

1. A buffering scheme that masks unpredictable I/O latency

2. A real-time safe implementation that is portable to mainstream operating systems

3. Design goals:

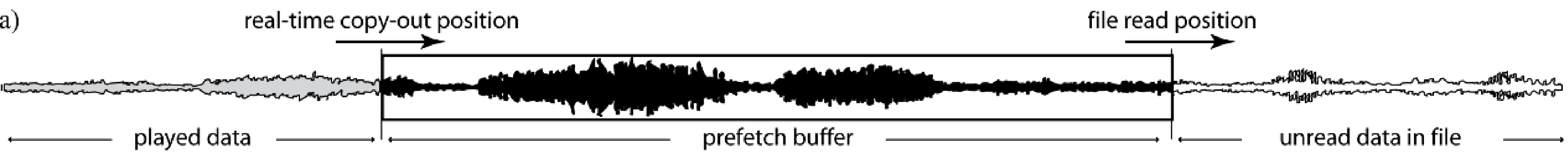
- Usable programming model for C/C++
- No busy-waiting
- Reliable/easy to reason about correctness

Outline for Rest of Talk

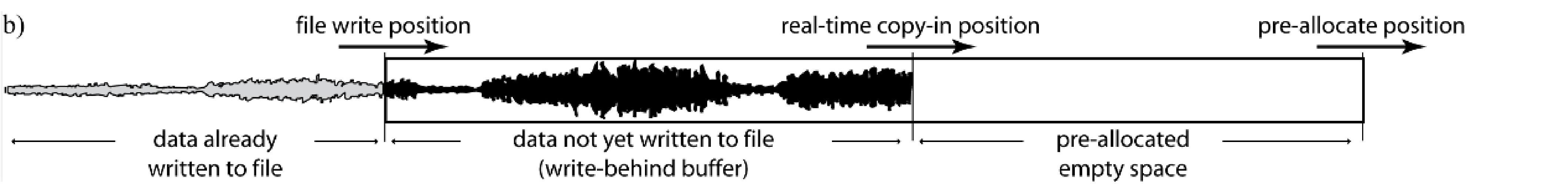
- Audio I/O and buffering model
- Message passing algorithm (high-level view)
- Implementation details
- Real-time-safe inter-thread communications using lock free queues
- Caveats, discussion

Audio I/O and Buffering Model

Playback:



Recording:



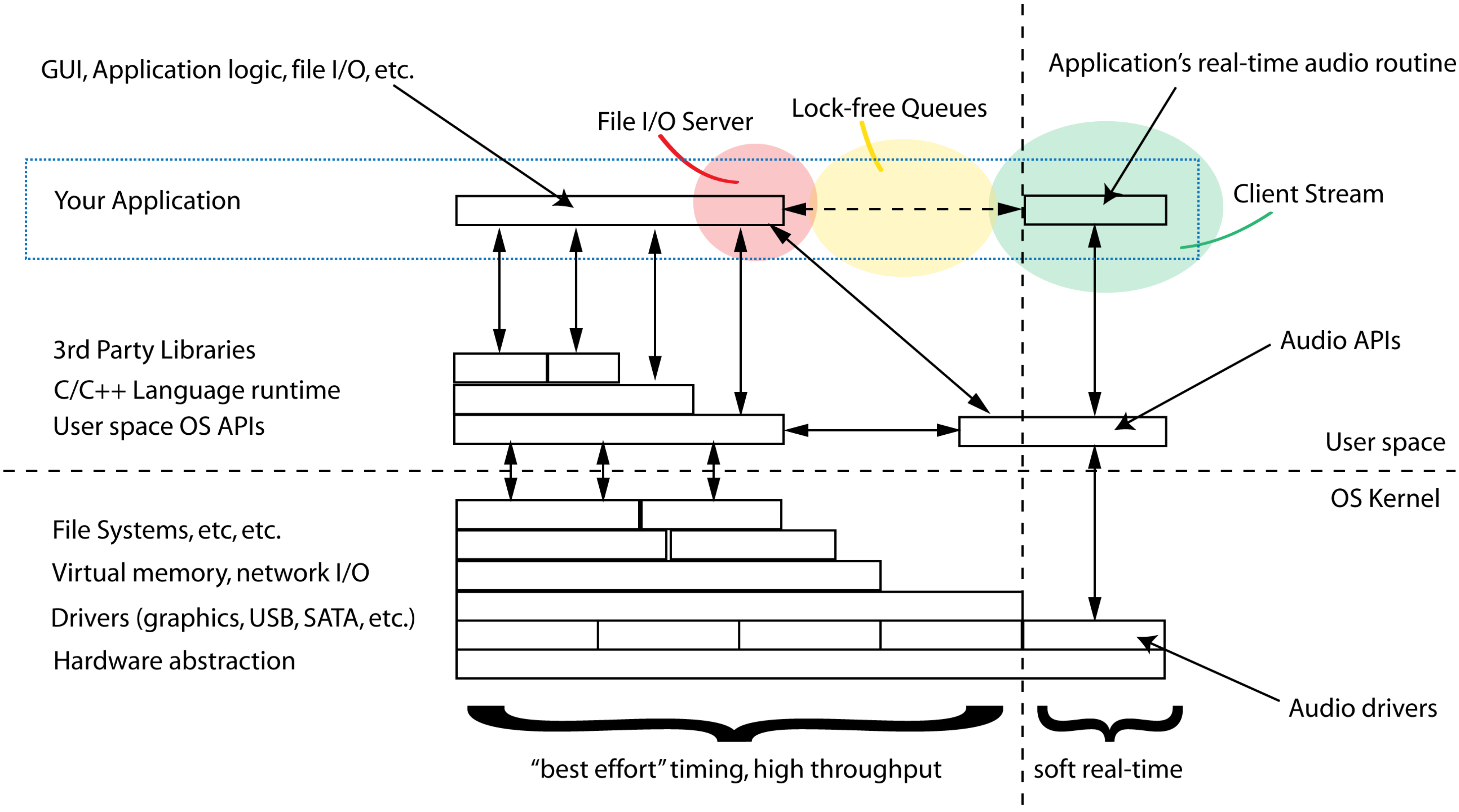
Streaming playback algorithm
(see animation)

Participants:
- Client Stream
- Server (performs non-real-time-safe operations)
- Requests
- Data Blocks
- Queues
- File Handles
- Threads

Process

Message Protocol (see later slide)

Threads in relation to application picture



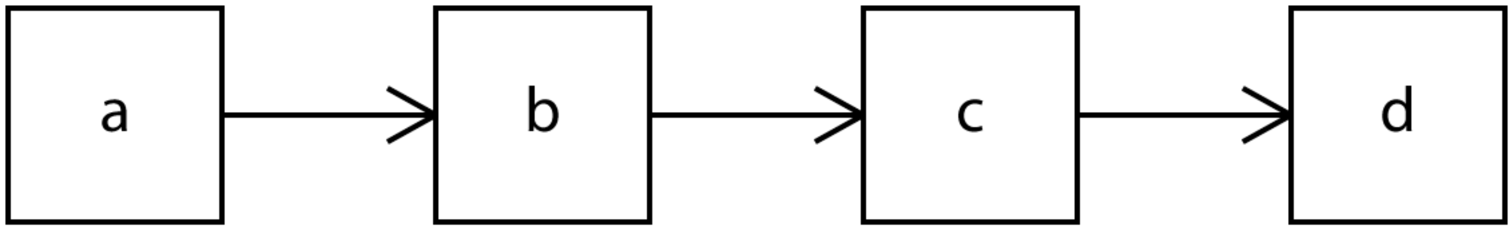
Note: Artist's impression only

Async File I/O Protocol Messages

```
OPEN_FILE (path, accessMode) ⇀ fileHandle | error

CLOSE_FILE fileHandle ⇀ ∘


READ_BLOCK (fileHandle, position) ⇀ dataBlock | error

RELEASE_READ_BLOCK (fileHandle, dataBlock) ⇀ ∘


ALLOCATE_WRITE_BLOCK (fileHandle, position)
                                     ⇀ dataBlock | error

COMMIT_MODIFIED_WRITE_BLOCK (fileHandle, dataBlock) ⇀ ∘

RELEASE_UNMODIFIED_WRITE_BLOCK (fileHandle, dataBlock) ⇀ ∘
```

Interlude: Linked Lists

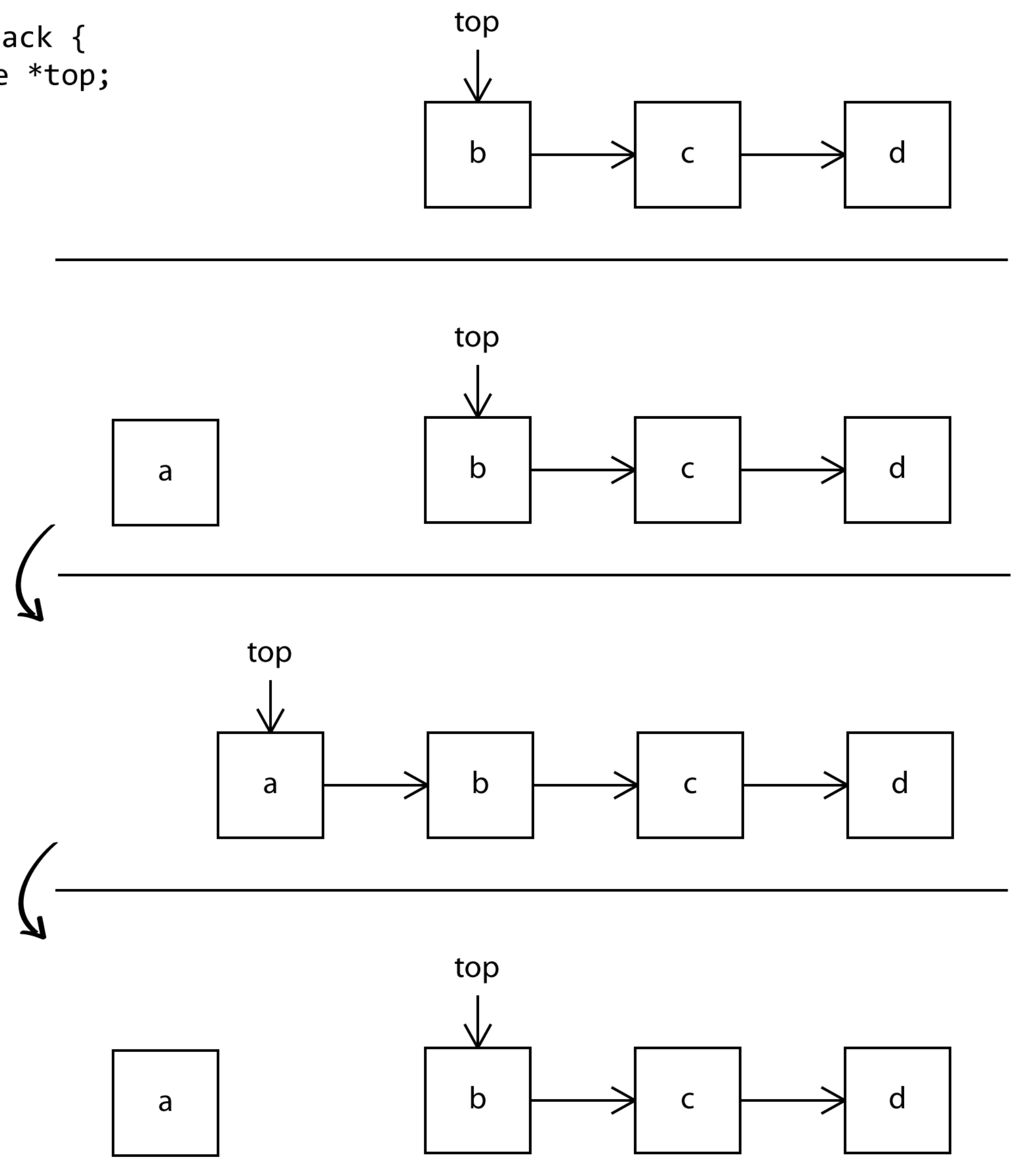
Linked List




```
struct Node {
     Node *next;
};

Node *a, *b, *c, *d;
a->next = b;
b->next = c;
c->next = d;
```

Last In First Out (LIFO) Stack

```
struct Stack {
     Node *top;
};
```

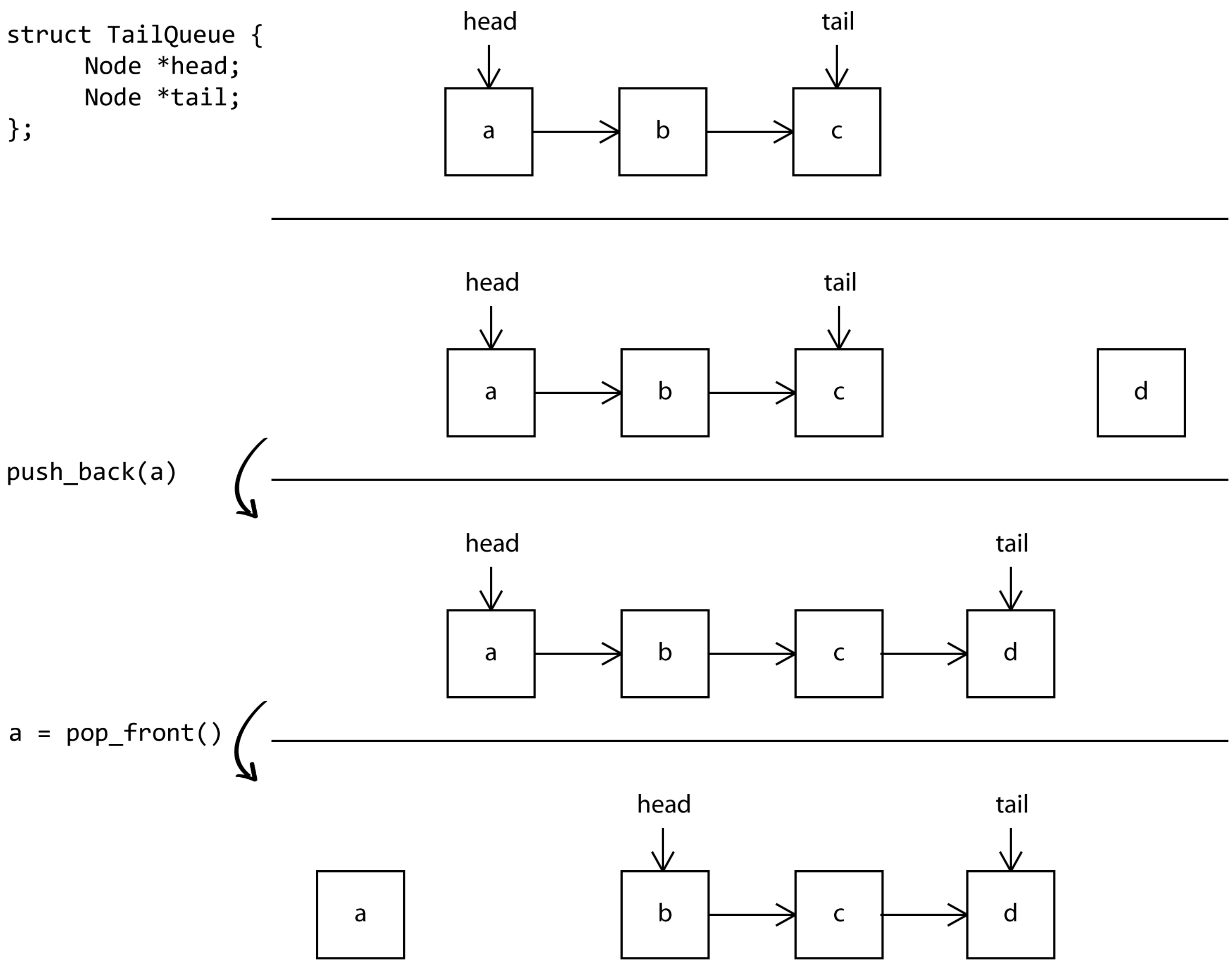



```
push(a)
```

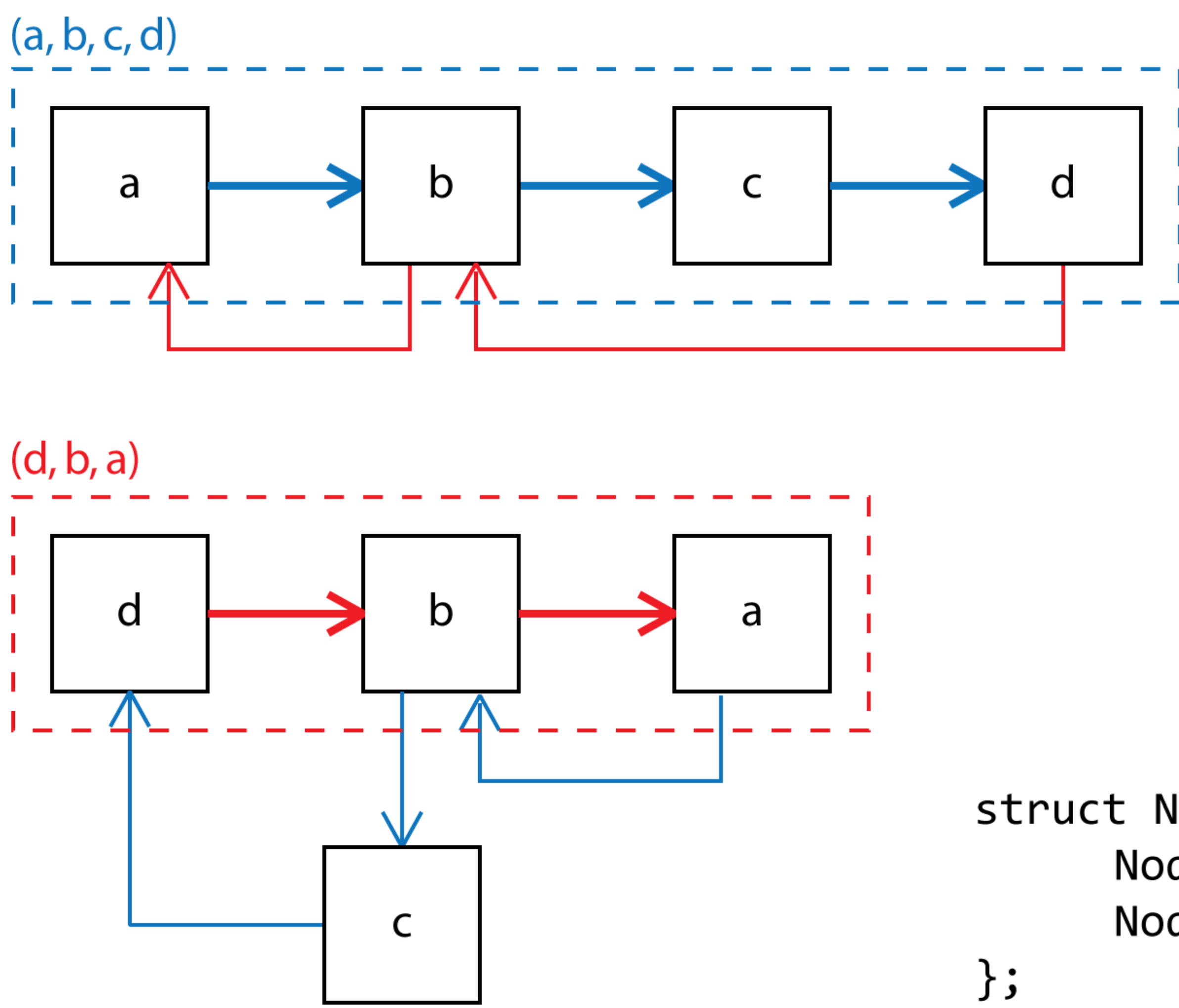
```
a = pop()
```

First In First Out (FIFO) “Tail Queue”

```
struct TailQueue {
      Node *head;
      Node *tail;
};
```



Two “next” pointers: same nodes, different lists




```
struct Node2 {
        Node *next_blue;
        Node *next_red;
};
```

(end interlude)

On to the implementation...

Requirements:

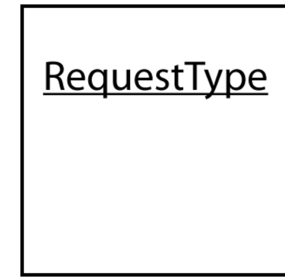
- Allocating, deallocating, sending requests and receiving replies must be real-time-safe.
- Server to process events in FIFO order. Clients may receive replies in any order.

Desiderata:

- Support immediate disposal of streaming state, without having to wait for pending replies.
- Real-time-safe construction and tear-down of stream state.
- Create and destroy streams in any thread, allow stream states to migrate between threads -- implies being able to send requests and receive replies from any thread.

Requests (messages) are Linked List Nodes

```
struct Request {
   Request *transitNext; // used by queues
   Request *clientNext; // used by Stream
   int requestType;
   int resultStatus;
   union {
      size_t clientInt;
      void *clientPtr;
   };
   union {
      // message-specific fields...
      DataBlock *dataBlock; // for example
   };
};
```



src/FileIoRequest.h

Aside: transit queues in the algorithm

Freelist (LIFO)

Server Queue (FIFO)

Result Queue (?)

Similar to linked lists but implemented using lock-free techniques – more later

Request Allocation/Deallocation

Requests are pre-allocated and stored in a freelist

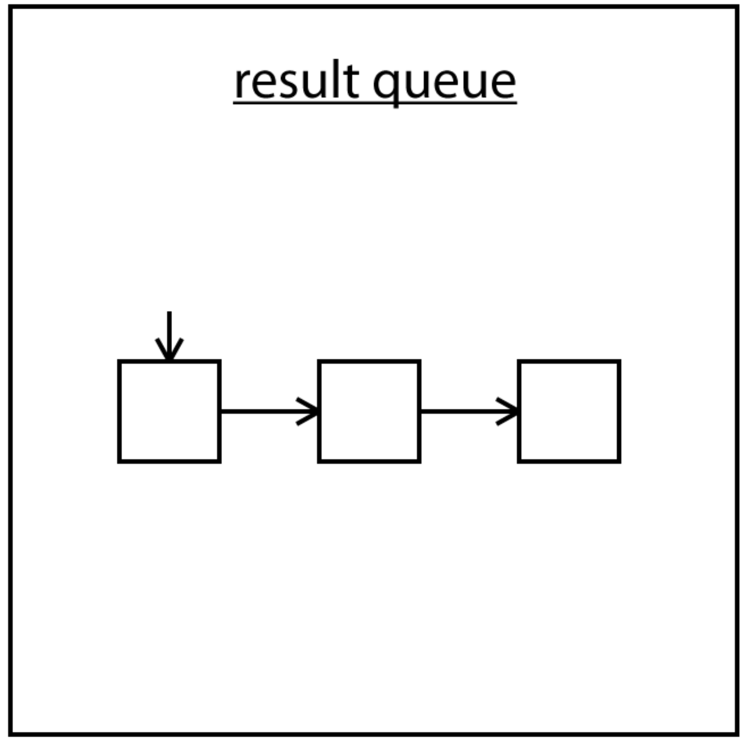
<u>To allocate a Request:</u>

pop a Request off the freelist

<u>To deallocate a Request:</u>

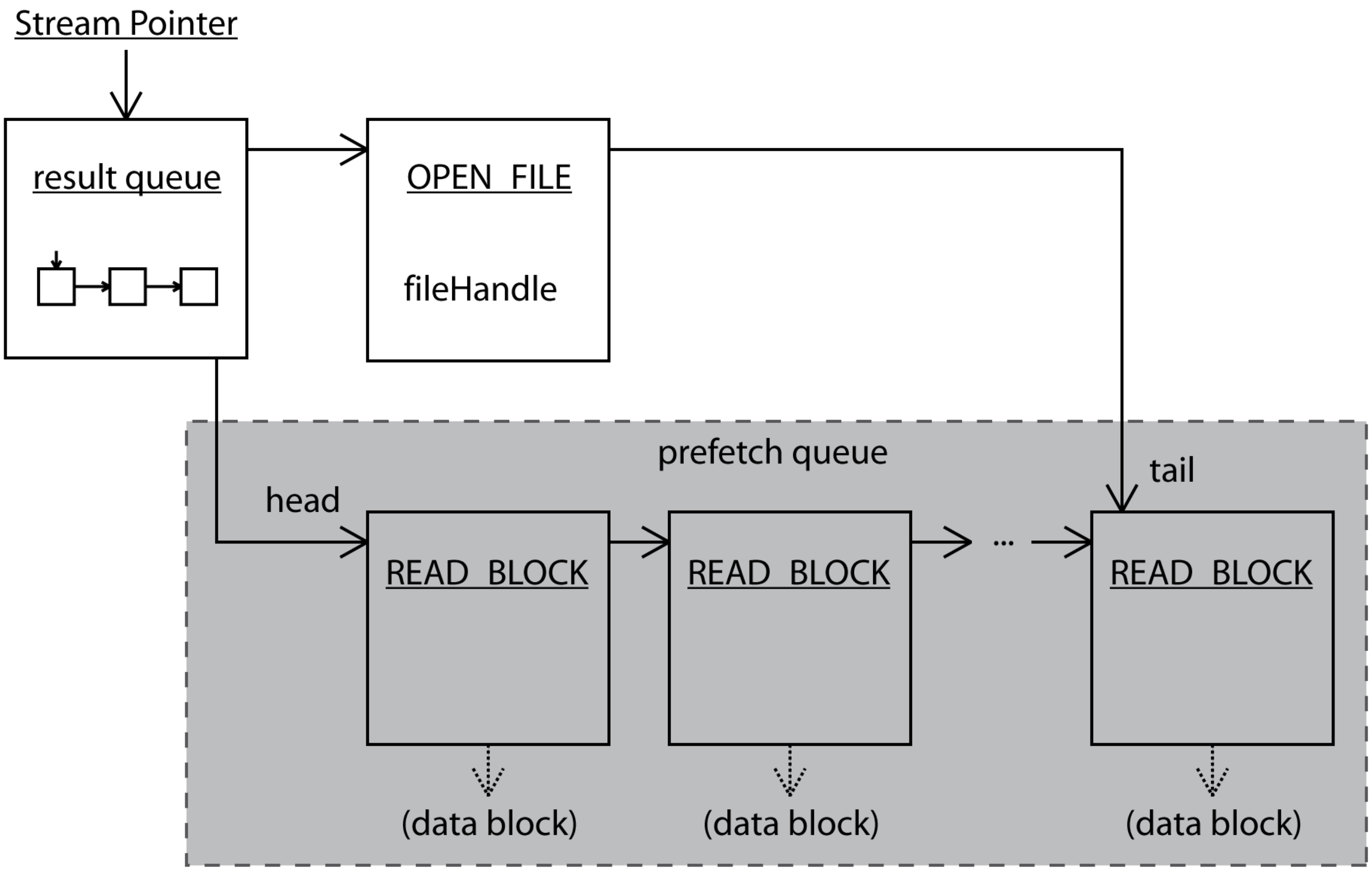
push a Request onto the freelist

Client Stream Result Queue is a Request



(Recall: Server pushes requests onto the result queue, Client pops them off.)

“Stream” objects are built out of linked Requests



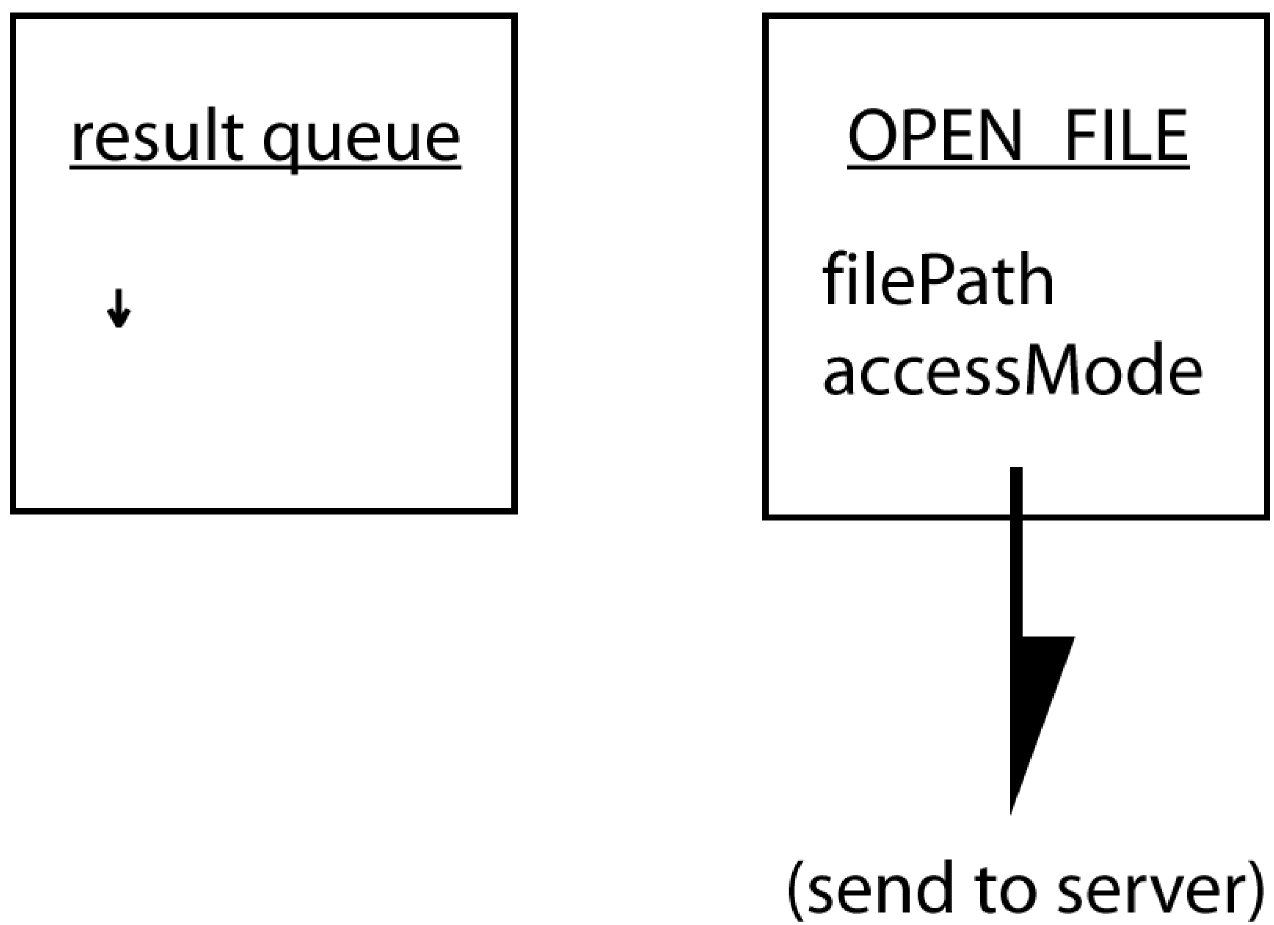
All stream state is stored in the Request nodes.

Client Stream Algorithms

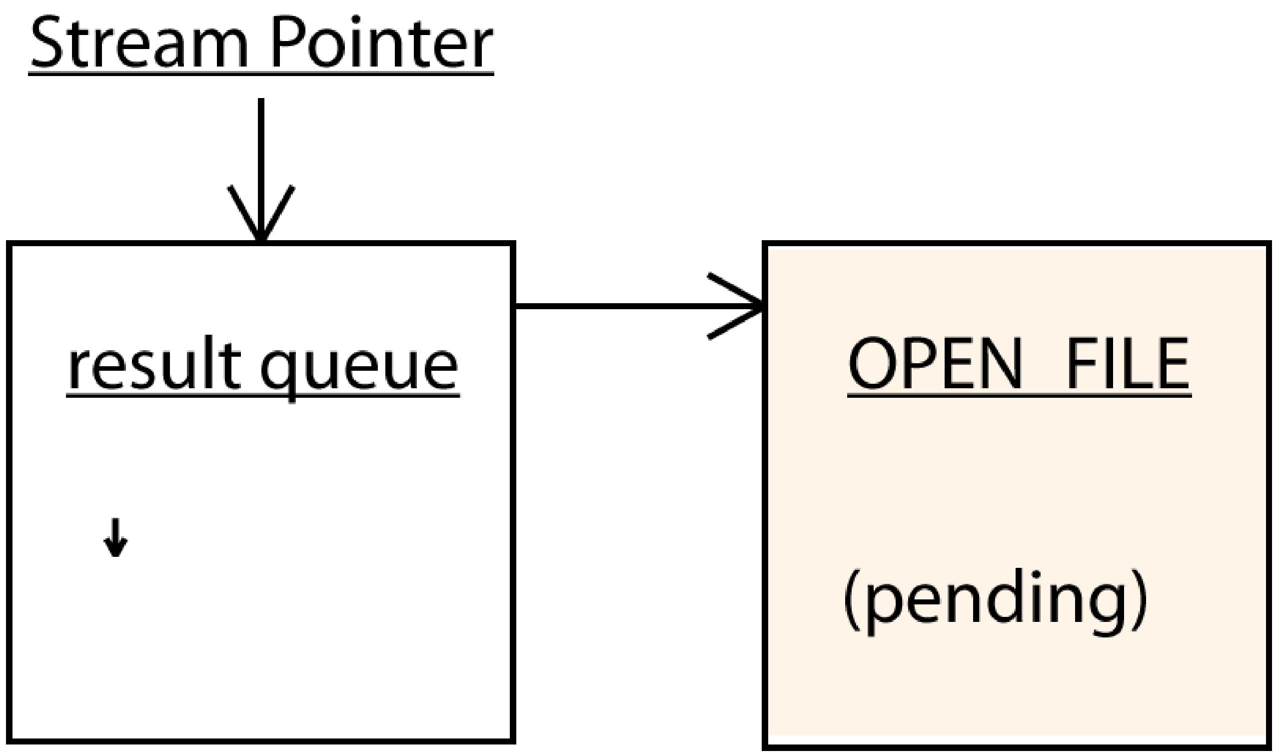
- Stream construction (bootstrapping)
- Prefetch queue maintenance (see paper)
- Stream destruction (highlighting async case)

Create Stream (bootstrapping)

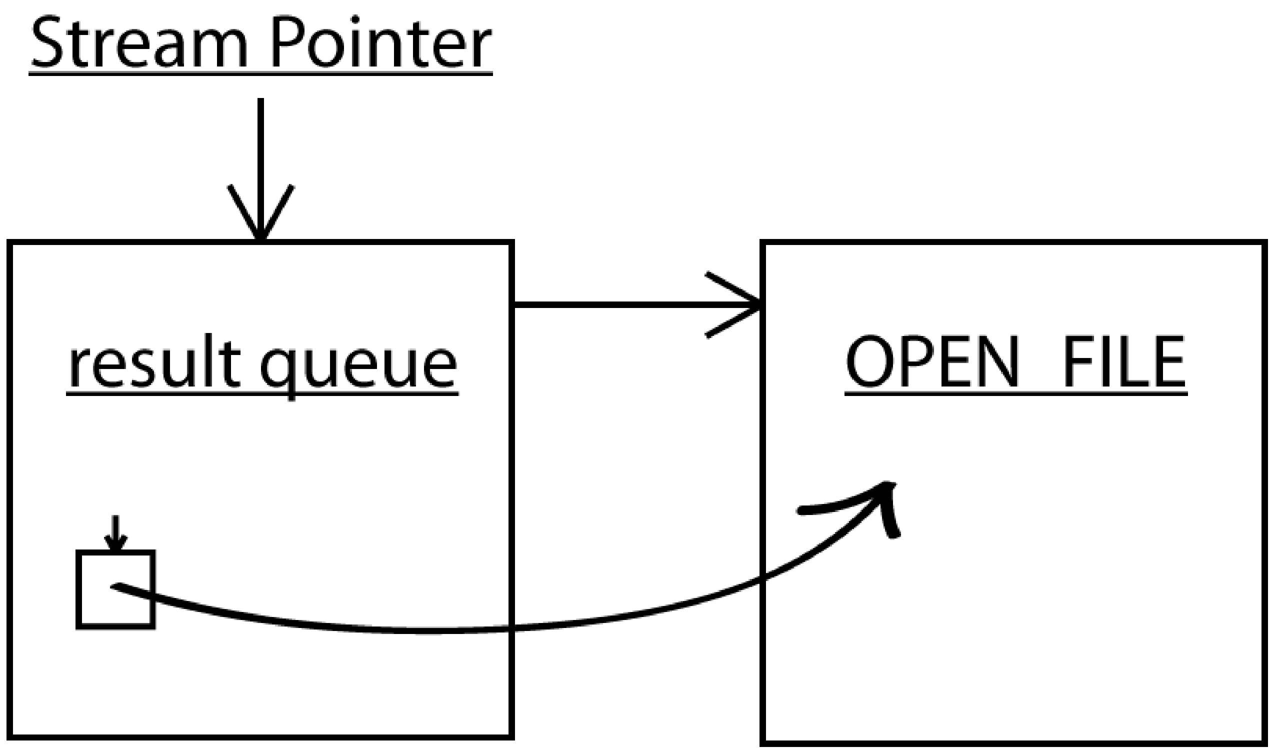
Create Stream 1/4



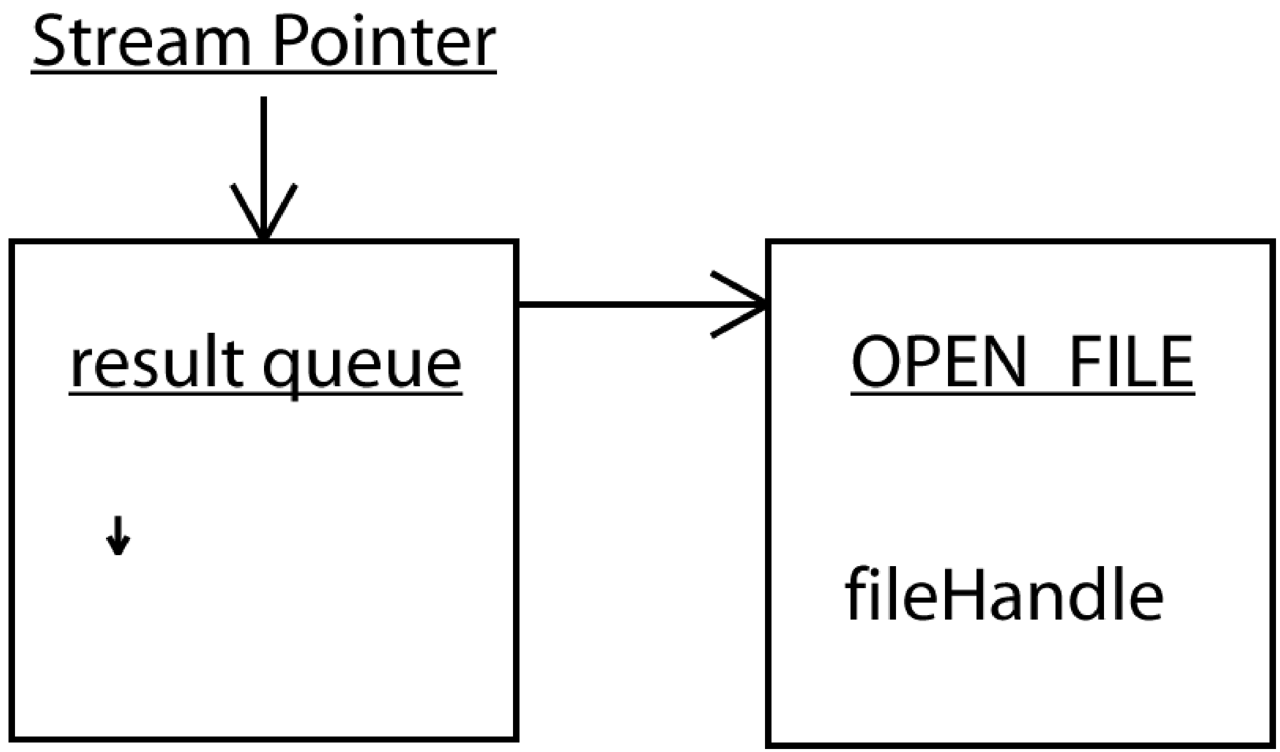
Create Stream 2/4



Create Stream 3/4

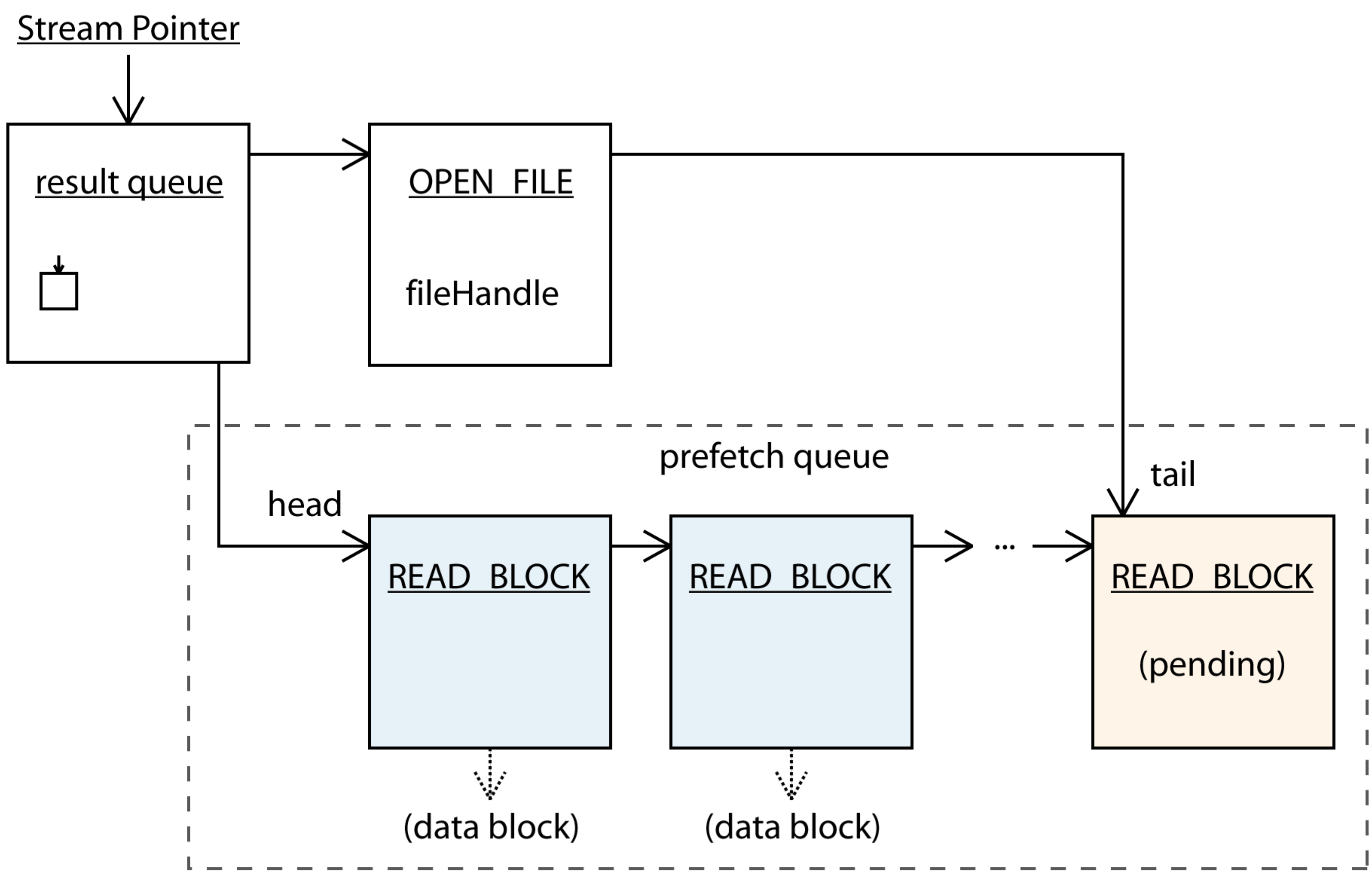


Create Stream 4/4

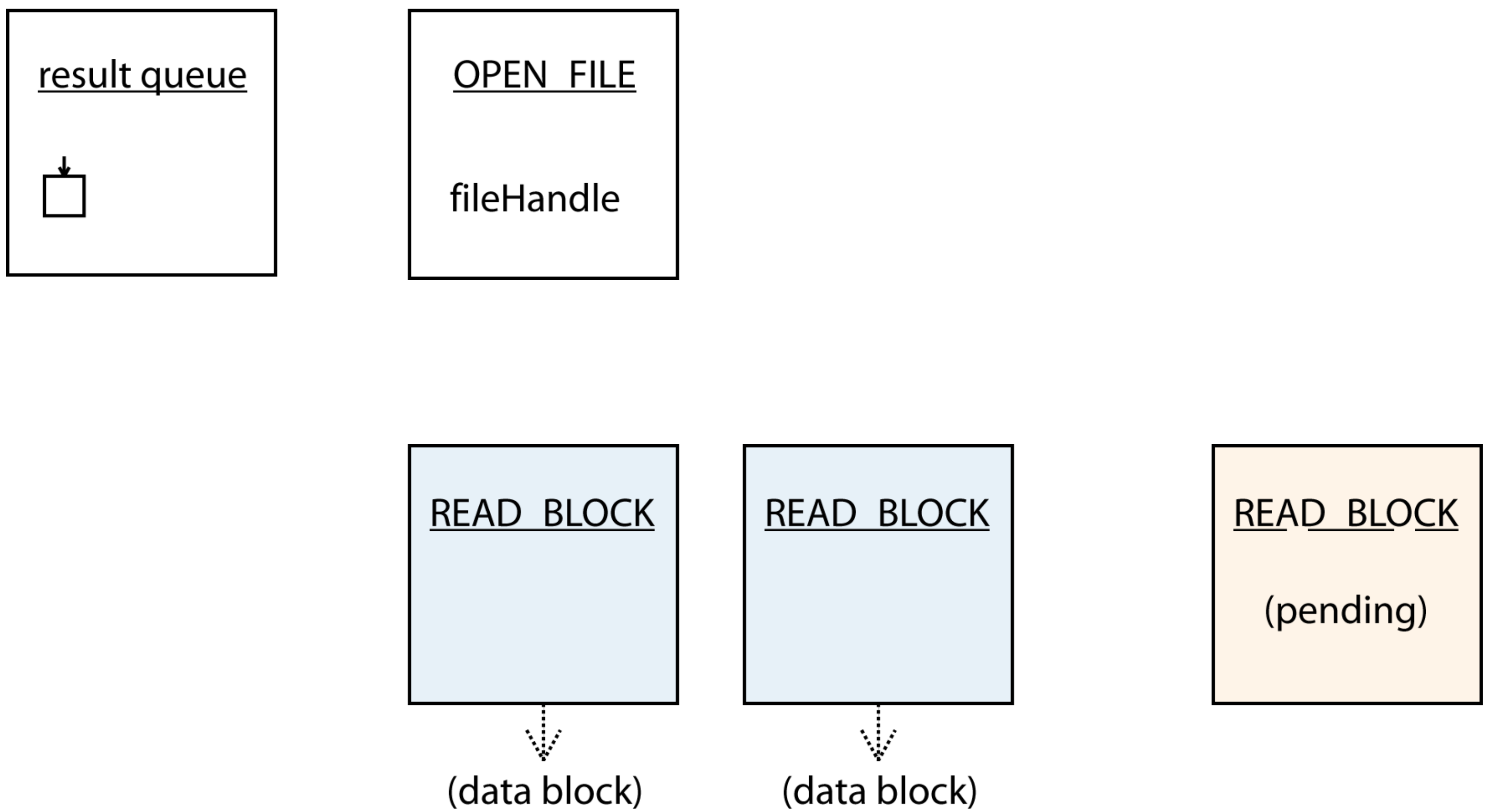


Destroy Stream (with pending requests)

Destroy Stream (with pending requests) 1/3

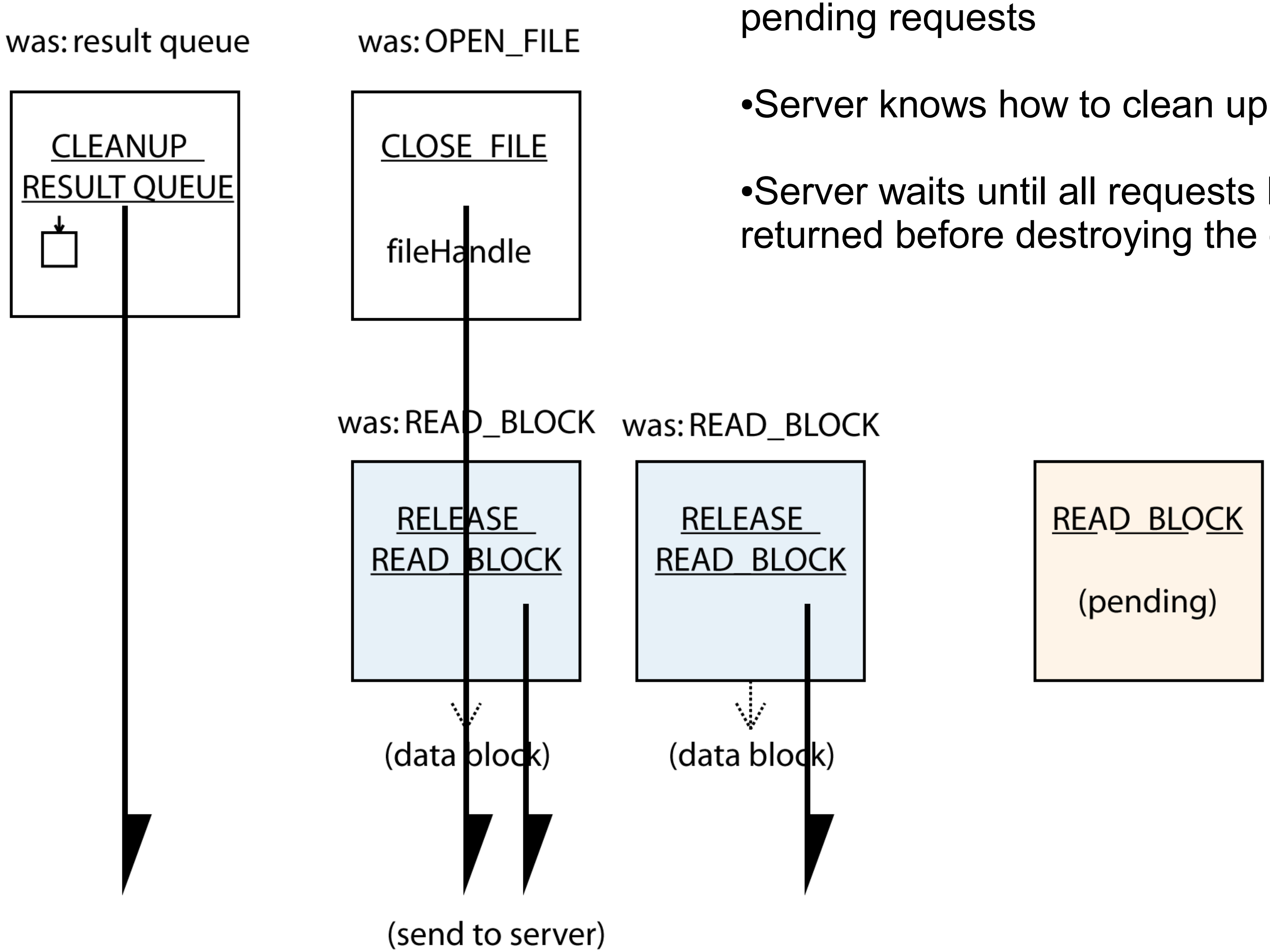


Destroy Stream (with pending requests) 2/3

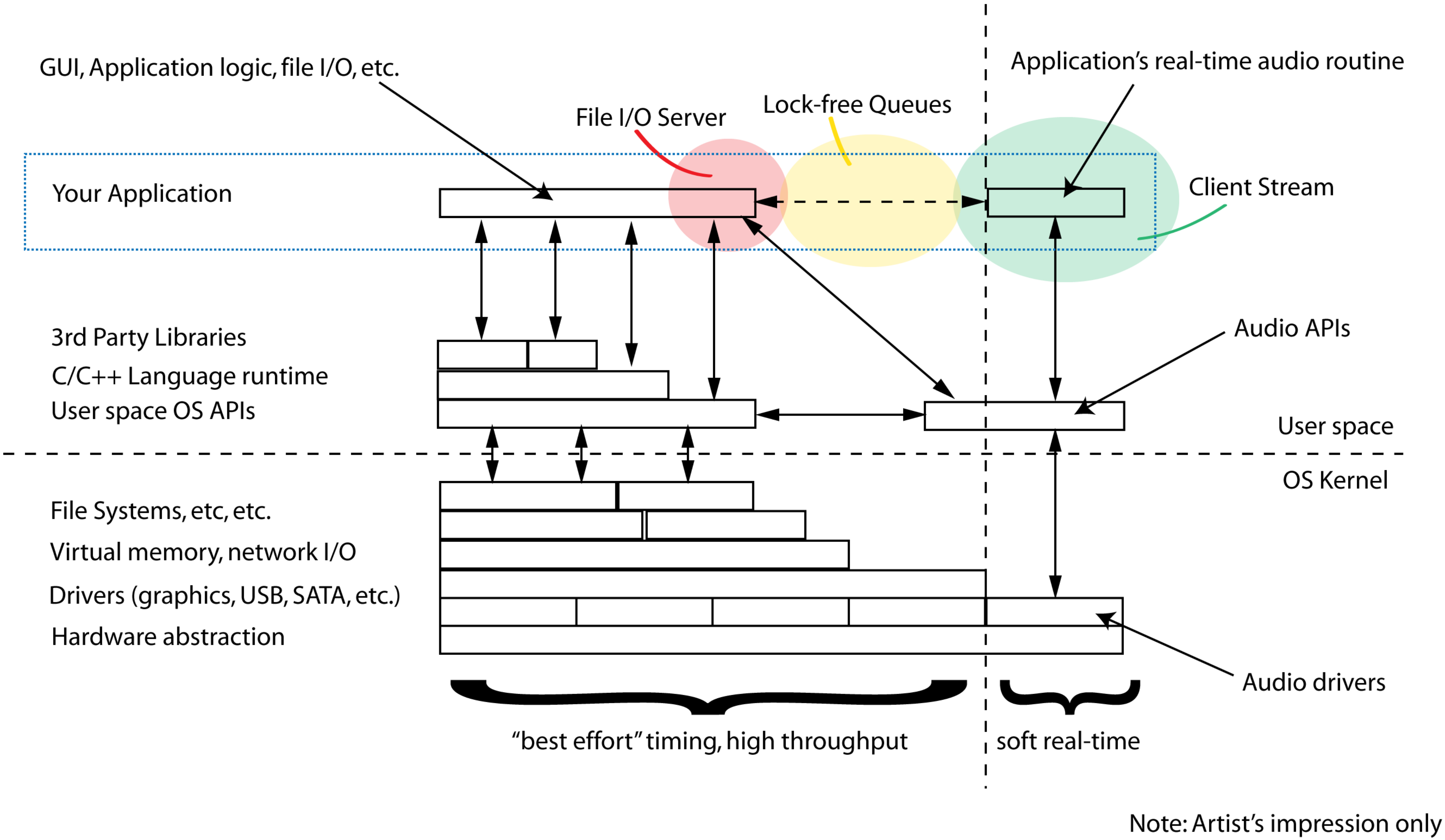


Destroy Stream (with pending requests) 3/3

- Result queue maintains a count of pending requests
- Server knows how to clean up requests
- Server waits until all requests have been returned before destroying the queue



How to make the queues real-time safe?

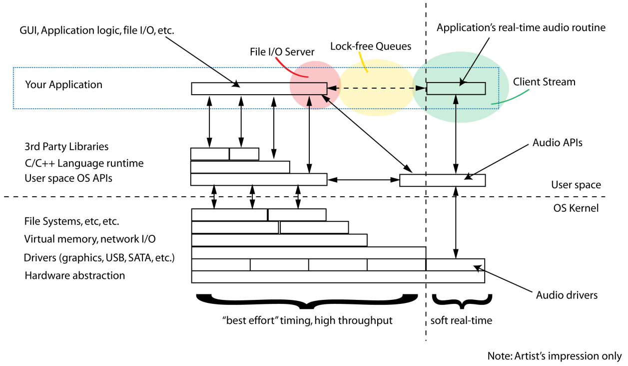

GUI, Application logic, file I/O, etc.
File I/O Server
Lock-free Queues
Application's real-time audio routine
Your Application
Client Stream
3rd Party Libraries
C/C++ Language runtime
User space OS APIs
Audio APIs
User space
OS Kernel
File Systems, etc, etc.
Virtual memory, network I/O
Drivers (graphics, USB, SATA, etc.)
Hardware abstraction
Audio drivers
"best effort" timing, high throughput
soft real-time
Note: Artist's impression only

Definition: A **lock-free algorithm** has the property that at least one thread makes global progress at each execution step.

E.g. if two or more threads concurrently push or pop from a lock-free queue, at least one makes progress.

Assuming bounded contention we get time-bounded completion of all threads. Has been shown to be provably real-time safe (on uniprocessor see Anderson et. al. 1997, waves hands about multicore).

Main advantage over mutexes: avoids priority inversion.



Use Lock-Free Queues for RT/NRT interface

Freelist: preallocate messages and store them in a stack (MPMC)

Server queue: pop-all FIFO (MPSC)

Client result queue: pop-all relaxed order (SPSC)

Lock-Free queue algorithms: see paper

Implementation based on “IBM Freelist”, atomic pop-all using XCHG, stack reversing.

- Single node struct for each message (simple, similar to simple linked list).
- Result queue can be embedded in request (the queue is just two pointers, no large/bounded ringbuffers)

https://github.com/RossBencina/QueueWorld

+ mintomic for portable C++ atomic ops

```
// lock-free pop-all stack:
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);
}
```

Recap on stated requirements...

Recap 1/2: Interlocking concerns:

1. A buffering scheme that masks unpredictable I/O latency

2. A real-time safe implementation that is portable to mainstream operating systems

3. Design goals:

- Usable programming model for C/C++
- No busy-waiting
- Reliable/easy to reason about correctness

Recap 2/2: Requirements:

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

Desiderata:

- Support immediate disposal of streaming state, without having to wait for pending replies.
- Real-time-safe construction and tear-down of stream state.
- Create and destroy streams in any thread, allow stream states to migrate between threads -- implies being able to send requests and receive replies from any thread.

Caveats

As presented: simplistic synchronous server

problem: server won't deal well with multiple streams (seek floods queue, need deadlines)

problem: haven't described how to parse sound file headers. Easy, low-performance: do it synchronously in server using libsndfile; Hard high-performance: do it asynchronously in server.

Not ideal: Seeking and stream tear-down is O(N) (each request in the prefetch queue has to be processed individually by the client).

Various ways of “improving” the protocol:
- elide READ and RELEASE messages
- bulk operations, e.g. READ-N message
- delegate some stream operations to server: move prefetch queue tear-down to server

Warning:

The queues used here are not strictly “wait-free.”
May not be appropriate for high-contention scenarios (i.e. many concurrent threads).
Further research needed.

Future Work

This talk has focused on the client/server interface. It turns out that implementing a high performance server in this style is a lot more work.

In progress: Async interface to native file I/O, caching and sharing file handles and data blocks among multiple clients, prioritised and cancellable requests, caching, data format conversion, etc.

Generalise beyond File I/O to any block based async server (e.g. Network, FFT processing, etc.)

Questions?