Neobuffer: Cross-Process Channels in Rust

Bernardo Meurer

June 28th, 2019

Standard Cognition

whoami

- Bernardo Meurer (`lovesegfault`)
- Systems Engineer @ Standard Cognition http://standard.ai
- `bernardo@standard.ai`
- http://lovesegfault.com/

Outline

- Motivation
- Requirements
- Alternatives
- Neobuffer
  - Components
  - Ringbuffer
  - Sink
  - Source
  - Quirks
  - Results
  - Future work
- Questions

whoarewe(?!)

- Standard Cognition
- SF + Tokyo
- Autonomous Checkout
- Python + Rust
- Many interesting problems to solve
- We’re hiring!

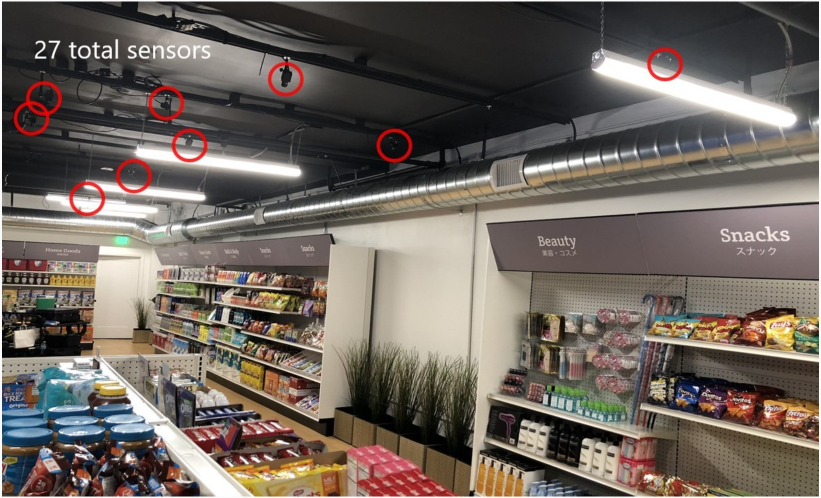
[Sc]

STANDARD COGNITION

Motivation

- We deal with a many cameras.

Motivation


27 total sensors
Home Goods
Snacks
Beauty
美容・コスメ
Snacks
スナック

Motivation

- We deal with a many cameras.
- We do experiments very frequently.

Motivation

Motivation

- We deal with a many cameras.
- We do experiments very frequently.
- How to multiplex the video streams?

Motivation

- We deal with a many cameras.
- We do experiments very frequently.
- How to multiplex the video streams?
- How to unify the API for controlling distinct cameras?

Sensord/Camd

- We still haven’t quite decided on a name…

Sensord/Camd

- We still haven’t quite decided on a name…
- Camera daemon

Sensord/Camd

- We still haven’t quite decided on a name...
- Camera daemon
- Clients use ZMQ to communicate with it

Sensord/Camd

- We still haven’t quite decided on a name…
- Camera daemon
- Clients use ZMQ to communicate with it
- Has client libs for Rust, and Python (PyO3)

Sensord/Camd

- We still haven't quite decided on a name...
- Camera daemon
- Clients use ZMQ to communicate with it
- Has client libs for Rust, and Python (PyO3)
- Reasonably large Rust project

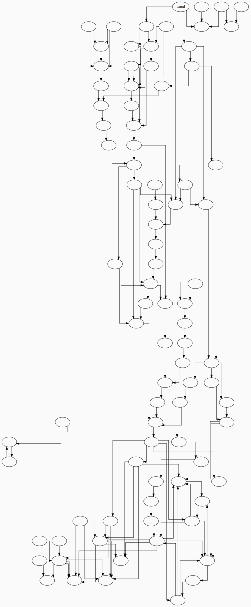
Sensord/Camd

```
$ tokei ./sensord
-------------------------------------------------------------------------------
 Language            Files        Lines         Code     Comments       Blanks
-------------------------------------------------------------------------------
 C Header                2            4            4            0            0
 GLSL                    2           63           54            1            8
 Markdown                1           29           29            0            0
 Nix                     1           44           34            0           10
 Python                  1           28           17            4            7
 Rust                   47         9391         7457          880         1054
 Plain Text              1           13           13            0            0
 TOML                   14          274          240            0           34
-------------------------------------------------------------------------------
 Total                  69         9846         7848          885         1113
-------------------------------------------------------------------------------
```

Sensord/Camd

- We still haven’t quite decided on a name...
- Camera daemon
- Clients use ZMQ to communicate with it
- Has client libs for Rust, and Python (PyO3)
- Reasonably large Rust project
- A small part of a huge system

Sensord/Camd



Sensord/Camd

- We still haven't quite decided on a name...
- Camera daemon
- Clients use ZMQ to communicate with it
- Has client libs for Rust, and Python (PyO3)
- Reasonably large Rust project
- A small part of a huge system
- One challenge was clear from inception...

Sensord/Camd

How to connect `camd` to the client processes?

Channel X: Requirements

- Fast

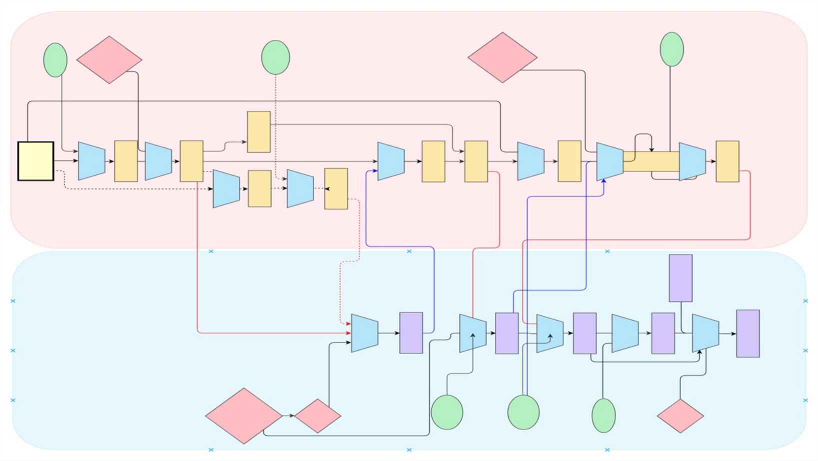
Channel X: Requirements

- Fast
- Lock-Free

Channel X: Requirements

- Fast
- Lock-Free
- Flexible

Channel X: Requirements



Channel X: Requirements

- Fast
- Lock-Free
- Flexible
- Cross-thread

Channel X: Requirements

- Fast
- Lock-Free
- Flexible
- Cross-thread
- Cross-process

Channel X: Requirements

- Fast
- Lock-Free
- Flexible
- Cross-thread
- Cross-process
- Safe

Channel X: Requirements

- Fast
- Lock-Free
- Flexible
- Cross-thread
- Cross-process
- Safe
- Multi-consumer

Channel X: Who could it be?

- stdlib channels?

Channel X: Who could it be?

- stdlib channels?
- `crossbeam`?

Channel X: Who could it be?

- stdlib channels?
- `crossbeam`?
- `rb`?

Channel X: Who could it be?

- stdlib channels?
- `crossbeam`?
- `rb`?
- `ringbuf`?

Channel X: Neobuffer

Neobuffer

- Written from scratch
- 100% Rust
- Collaboration with `ekleog` (Leo Gaspard) and `nagisa` (Simonas Kaslauskas)
- Based on discussions with `eddyb` and `amanieu`

Neobuffer: Components

- Common traits for process-level `Sync` and `Send`
  - `interprocess-traits` — `ProcSync` and `ProcSend`
  - github.com/standard-ai/interprocess-traits

Neobuffer: Components

- Common traits for process-level `Sync` and `Send`
  - `interprocess-traits` — `ProcSync` and `ProcSend`
  - github.com/standard-ai/interprocess-traits
- Shared memory for cross-process data sharing
  - `shmem` — An ergonomic wrapper around shared memory for Rust
  - github.com/standard-ai/shmem

Neobuffer: Components

- Common traits for process-level `Sync` and `Send`
    - `interprocess-traits` — `ProcSync` and `ProcSend`
    - github.com/standard-ai/interprocess-traits
- Shared memory for cross-process data sharing
    - `shmem` — An ergonomic wrapper around shared memory for Rust
    - github.com/standard-ai/shmem
- Unix Domain Sockets for sending file descriptors across processes
    - `sendfd` — Utility crate that simplifies `fd` sharing.
    - github.com/standard-ai/sendfd

Neobuffer: Components

- Common traits for process-level `Sync` and `Send`
  - `interprocess-traits` — `ProcSync` and `ProcSend`
  - github.com/standard-ai/interprocess-traits
- Shared memory for cross-process data sharing
  - `shmem` — An ergonomic wrapper around shared memory for Rust
  - github.com/standard-ai/shmem
- Unix Domain Sockets for sending file descriptors across processes
  - `sendfd` — Utility crate that simplifies `fd` sharing.
  - github.com/standard-ai/sendfd
- Ringbuffer for the underlying data structure
  - Atomics for the indexes
  - Careful application of memory orderings to guarantee consistency
  - Safe wrap around
  - github.com/standard-ai/neobuffer

Neobuffer: Ringbuffer, Sink, and Source

```rust
/// A shared-memory ring buffer that contains items of type `T`, has one writer
/// and `NbReaders` readers that each read all the items the writer writes.
pub struct RingBuffer<T, NbReaders> {
    /// The shared memory region
    data: Shared<c_void>,

    /// The maximum number of elements this ring buffer can contain at a time.
    size: u64,

    /// Whether the one writer of this ring buffer has already been given out.
    writer_given: AtomicBool,

    /// The number of readers that have already been given out.
    readers_given: AtomicUsize,

    /// Phantom argument to keep type arguments alive.
    phantom: PhantomData<(T, NbReaders)>,
}
```

Neobuffer: Ringbuffer, Sink, and Source

- `Shared<T>` just means `T` is in shared memory

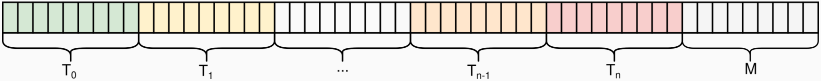
Neobuffer: Ringbuffer, Sink, and Source

- `Shared<T>` just means `T` is in shared memory
- `Shared<c_void>` is a special case for data not sized

Neobuffer: Ringbuffer, Sink, and Source

- `Shared<T>` just means `T` is in shared memory
- `Shared<c_void>` is a special case for data not sized
- But what exactly is in `Shared<c_void>`?

Neobuffer: Ringbuffer, Sink, and Source



Neobuffer: Ringbuffer, Sink, and Source

- `Shared<T>` just means `T` is in shared memory
- `Shared<c_void>` is a special case for data not sized
- But what exactly is in `Shared<c_void>`?
- *M* is the Ringbuffer metadata, and the magic smoke of Neobuffer

Neobuffer: Ringbuffer, Sink, and Source

```
/// The metadata for the data stored in the `Shared`.
///
/// The actual data in the ringbuffer precedes this metadata, so that it can be
/// page-aligned. Elements between the minimal element of `reader_steps` and
/// `writer_step` (modulo `Size`) are the only initialized ones. The safety of
/// this relies on the fact that things in a `Shared` are guaranteed to not be
/// dropped, and to the dropping operations being manually handled.
#[repr(C)]
struct Metadata {
    /// Number of readers allocated in the metadata area (following this struct).
    ///
    /// This is immutable.
    reader_count: usize,

    /// The number of items that have ever been written. Monotonically
    /// increasing. `u64` is used here because `u32` might not be enough to
    /// handle a reasonable number of ring buffer additions.  However, all uses
    /// that are bounded by `Size` can safely assume it fits in `usize`.
    writer_step: AtomicU64,

    /// The number of outstanding references to this shared memory region.
    ///
    /// This knowledge is used to know when the remaining elements should be
    /// dropped.
    references: AtomicU64,
}
```

Neobuffer: Ringbuffer, Sink, and Source

```rust
impl<T, R> RingBuffer<T, R>
where
    // Ts are not created/inserted/shared here, so T does not need any bound here.
    R: Unsigned + NonZero,
{
    pub fn new(size: usize) -> Result<RingBuffer<T, R>, shmem::Error> {...}
    pub fn get_sink(&self) -> Result<Sink<T>, Error> {...}
    pub fn get_source(&self) -> Result<Source<T>, Error> {...}
}
```

Neobuffer: Ringbuffer, Sink, and Source

```
/// A sink that can send data into a `RingBuffer`.
pub struct Sink<T> {
    /// Shared pointer to the data.
    data: Shared<c_void>,

    /// Number of `T` that `data` has been allocated with.
    size: u64,

    /// Local copy of `data.writer_step`, that is not atomic for performance.
    writer_step: u64,

    /// Local helper that contains a value guaranteed to be lower to or equal
    /// to all values in `data.reader_steps`
    minimum_reader_step: Cell<u64>,

    /// Phantom argument to keep type arguments alive.
    phantom: PhantomData<T>,
}
```

Neobuffer: Ringbuffer, Sink, and Source

```
impl<T> Sink<T> {
    pub fn push(&mut self, item: T) -> nb::Result<(), Error> {...}
}
```

Neobuffer: Ringbuffer, Sink, and Source

```rust
/// A sink that can read data from a `RingBuffer`.
pub struct Source<T> {
    /// Shared pointer to the data.
    data: Shared<c_void>,

    /// Size of the ring buffer this is linked with.
    size: u64,

    /// Identifier of this reader in the `data.reader_steps` array.
    reader_id: usize,

    /// Local copy of `data.reader_steps[reader_id]`, for performance.
    reader_step: u64,

    /// Under-approximation of `data.writer_step`, local for performance.
    minimum_writer_step: Cell<u64>,

    /// Phantom argument to keep type arguments alive.
    phantom: PhantomData<T>,
}
```

Neobuffer: Ringbuffer, Sink, and Source

```rust
impl<T> Source<T> {
    pub fn available_size(&self) -> usize {...}

    /// Advance the reader discarding `n` lowest numbered elements in the buffer.
    ///
    /// This allows elements, which have been advanced past by all readers, to
    /// be overwritten.
    pub fn advance(&mut self, n: usize) -> nb::Result<(), Infallible> {...}

    /// Get a reference to `i`-th element stored in the buffer.
    ///
    /// Once the use of the reference is done, [`Source::advance`] must be
    /// called to advance the reader.
    pub fn get(&self, i: usize) -> nb::Result<&T, Infallible> {...}

    /// Handy combination of `get()` and `advance()`
    pub fn pop(&mut self) -> nb::Result<T, Infallible> {...}
}
```

Neobuffer: Quirks

- Consumption is local

Neobuffer: Quirks

- Consumption is local
- Relative indexing

Neobuffer: Quirks

- Consumption is local
- Relative indexing
- Only works on Linux

Neobuffer: Quirks

- Consumption is local
- Relative indexing
- Only works on Linux
- You have to be surgically careful with atomic orderings

Neobuffer: Results

```
one_writer_many_batching_readers_big   ... 10,693,940 ns/iter (+/- 931,225) = 1568 MB/s
one_writer_many_batching_readers_huge  ... 2,207,889 ns/iter (+/- 370,206) = 30395 MB/s
one_writer_many_batching_readers_small ... 9,169,666 ns/iter (+/- 411,746) = 7 MB/s
one_writer_many_readers_big            ... 7,919,200 ns/iter (+/- 584,713) = 2118 MB/s
one_writer_many_readers_huge           ... 2,219,540 ns/iter (+/- 186,404) = 30235 MB/s
one_writer_many_readers_small          ... 6,706,252 ns/iter (+/- 480,979) = 9 MB/s
one_writer_one_batching_reader_big     ... 3,522,033 ns/iter (+/- 18,138) = 4763 MB/s
one_writer_one_batching_reader_huge    ... 2,188,906 ns/iter (+/- 19,540) = 30658 MB/s
one_writer_one_batching_reader_small   ... 3,369,997 ns/iter (+/- 314,502) = 19 MB/s
one_writer_one_reader_big              ... 5,138,619 ns/iter (+/- 263,518) = 3264 MB/s
one_writer_one_reader_huge             ... 2,196,294 ns/iter (+/- 31,967) = 30555 MB/s
one_writer_one_reader_small            ... 3,213,515 ns/iter (+/- 360,083) = 20 MB/s
one_writer_two_batching_readers_big    ... 5,827,360 ns/iter (+/- 160,344) = 2879 MB/s
one_writer_two_batching_readers_huge   ... 2,201,807 ns/iter (+/- 14,876) = 30478 MB/s
one_writer_two_batching_readers_small  ... 4,877,466 ns/iter (+/- 184,255) = 13 MB/s
one_writer_two_readers_big             ... 5,310,644 ns/iter (+/- 149,770) = 3159 MB/s
one_writer_two_readers_huge            ... 2,195,695 ns/iter (+/- 91,573) = 30563 MB/s
one_writer_two_readers_small           ... 4,028,670 ns/iter (+/- 403,310) = 16 MB/s
```

Neobuffer: Results

Assuming 4 parallel readers and a buffer size of 8:

- 7 Mpps (1 byte packets)
  - 50Mpps with a buffer size of 2048

Neobuffer: Results

Assuming 4 parallel readers and a buffer size of 8:

- 7 Mpps (1 byte packets)
  - 50Mpps with a buffer size of 2048
- 300 Gbps (2MB packets)

Neobuffer: Future work

- Improve ergonomics around failure cases

Neobuffer: Future work

- Improve ergonomics around failure cases
- Split pushing into allocate + commit (remove a copy/clone)

Neobuffer: Future work

- Improve ergonomics around failure cases
- Split pushing into allocate + commit (remove a copy/clone)
- Swappable backend

Neobuffer: Future work

- Improve ergonomics around failure cases
- Split pushing into allocate + commit (remove a copy/clone)
- Swappable backend
- MPMC

Neobuffer: Future work

- Improve ergonomics around failure cases
- Split pushing into allocate + commit (remove a copy/clone)
- Swappable backend
- MPMC
- Futures

Neobuffer: Future work

- Improve ergonomics around failure cases
- Split pushing into allocate + commit (remove a copy/clone)
- Swappable backend
- MPMC
- Futures
- Global consumption

Summary

- Lock-free, fast, cross-process channel

Summary

- Lock-free, fast, cross-process channel
- Ringbuffer in shared memory

Summary

- Lock-free, fast, cross-process channel
- Ringbuffer in shared memory
- Public supporting libraries

Summary

- Lock-free, fast, cross-process channel
- Ringbuffer in shared memory
- Public supporting libraries
- Atomics to manage read/write indexes

Summary

- Lock-free, fast, cross-process channel
- Ringbuffer in shared memory
- Public supporting libraries
- Atomics to manage read/write indexes
- Safe interface for cross-process communication

Questions?

There will be no questions

Are there any questions?

Made with Free Software

This presentation was made using Free (as in Freedom) and Open Source software:

- Vim
- LaTeX + Beamer + `metropolis`
- Graphviz
- Inkscape
- Some dude on Fiverr